MASTER EN SCIENCES INFORMATIQUES

Conception d'un exercice intégré pour un cours de sécurité

Boogaerts, Yannick

Award date:
2009

Facultés Universitaires Notre-Dame de la Paix, Namur

Institut d'informatique.

Année académique 2008 - 2009

Conception d'un exercice intégré pour un cours de sécurité

Yannick Boogaerts

Mémoire présenté en vue de l'obtention du grade de licencié en informatique.

Résumé

Le cours de sécurité, dans le cadre d'une formation professionnelle destinée à des analystes-programmeurs juniors, pose des problèmes importants. Les techniques de sécurité sont complexes et généralement vécues par les étudiants comme une source de problèmes à la réalisation d'applications dont ils se passeraient bien.

Ce cours se doit de développer en plus des compétences transversales telles que l'acquisition d'une bonne logique de programmation et de la conception objet en Java.

Vu ce contexte et le temps relativement limité qu'il nous est imparti nous avons imaginé un exercice intégré alliant apprentissage théorique et pratique de programmation.

Par l'utilisation d'une pédagogie active, l'exercice développera chez les étudiants la compréhension des notions de base indispensables à l'appropriation des techniques présentes dans les API Java. Ils pourront aussi développer leur technique de travail et leur capacité d'autoformation, démarche indispensable pour la poursuite de leur apprentissage initié dans ce cours.

Mots clés : sécurité, java, pédagogie active, mise en situation

Abstract

Security course, as part of a vocational training for junior analyst programmers, poses significant problems. Security techniques are complex and generally experienced by students as a source of problems for the realization of application, problems that they'd like to avoid if they can.

This course must develop more cross-curricular competencies such as acquiring a good sense of programming and object designing in Java.

Given this background and the relatively limited time allocated, we imagine an integrated exercise combining learning theory and practice of programming.

By using an active pedagogy, exercise develop students' comprehension of basic concepts essential to ownership techniques presented in the Java API. They will also develop their technical work and their ability of self learning, prerequisites for further learning introduces in this course.

Keywords: security, java, active learning, development situation

Table des matières

Glossaire

Ant	est un projet open source de la fondation Apache écrit en Java qui vise le développement d'un logiciel d'automatisation des opérations répétitives tout au long du cycle de développement logiciel, à l'instar des logiciels « Make ».
Bac à sable	est un mécanisme qui permet l'exécution de logiciel(s) avec moins de risques au niveau du système d'exploitation. Ces derniers sont souvent utilisés pour exécuter du code non testé ou de provenance douteuse. Le terme anglais est *SandBox*.
Classe abstraite	est une classe dont l'implémentation n'est pas complète et qui n'est pas instanciable. Elle sert de base à d'autres classes dérivées (héritées).
Codesource	dans le contexte de la sécurité, le codesource représente l'origine du chargement du code. Cette origine peut être garantie grâce à un certificat.
Design Pattern	est un concept destiné à résoudre les problèmes récurrents suivant le paradigme objet. Il décrit un modèle de conception général. On peut considérer un *Design Pattern* comme une formalisation de bonnes pratiques, ce qui signifie qu'on privilégie les solutions éprouvées.
Fabrique	c'est un *Design Pattern* qui a pour rôle l'instanciation d'objets divers dont le type n'est pas prédéfini : les objets sont créés dynamiquement en fonction des paramètres passés à la fabrique.
Fabrique abstraite	encapsule un groupe de fabriques ayant une thématique commune
Fichier « jar »	est un fichier « zip » utilisé pour distribuer un ensemble de classes Java. Ce format est utilisé pour stocker les définitions des classes ainsi que des métadonnées constituant l'ensemble d'un programme.

Keystore	est un fichier protégé par mot de passe, qui peut contenir différentes clés et certificats. Il est également appelé le magasin de clés. C'est aussi une classe Java dont les instances ont la capacité de stocker en mémoire une collection de clés et de certificats.
Keytool	est un utilitaire fournit avec la plateforme Java qui permet de gérer les clés et les certificats.
Méthode statique	est une méthode qui peut être appelée même sans avoir instancié la classe. Une méthode statique ne peut accéder qu'à des attributs et méthodes statiques.
MVC	est un *Design Pattern* qui organise l'interface homme-machine (IHM) d'une application logicielle. Il divise l'IHM en un modèle (modèle de données), une vue (présentation, interface utilisateur) et un contrôleur (logique de contrôle, gestion des événements, synchronisation), chacun ayant un rôle précis dans l'interface.
OpenSSL	est une boite à outils de chiffrement comportant deux bibliothèques (une de cryptographie générale et une implémentant le protocole SSL), ainsi qu'une commande en ligne.
Package	ensemble de classes Java appartenant au même domaine de nom.
PAM	*Pluggable Authentication Module* est un mécanisme flexible d'authentification des utilisateurs.
RFC	les *Requests For Comments* (RFC), littéralement "demande de commentaires", sont une série numérotée de documents officiels décrivant les aspects techniques d'Internet. Peu de RFC sont des standards, mais tous les standards d'Internet sont des RFC.
SecurityManager	est une classe Java qui permet aux applications d'implémenter une politique de sécurité.
Signature électronique	une donnée sous forme électronique qui est jointe ou liée logiquement à d'autres données électroniques et qui sert de méthode d'authentification.

Socket	est une interface logicielle permettant au développeur d'exploiter facilement et de manière uniforme les services d'un protocole réseau.
SSL	est un sigle qui représente un protocole de sécurisation des échanges sur Internet, devenu *Transport Layer Security* (TLS) en 2001.
UIT	l'Union Internationale des Télécommunications est l'institution spécialisée des Nations Unies pour les technologies de l'information et de la communication. L'UIT a son siège à Genève, Suisse, et compte 191 états membres et plus de 700 membres de secteurs et associés.

Introduction

Le propos de ce mémoire est la conception d'un exercice intégré servant de base à l'apprentissage concret des techniques de sécurisation d'une application Java.

Cet exercice devra :

- Etre adapté à un public d'adultes participant à une formation professionnelle de programmeur Java.
- Permettre les manipulations nécessaires à l'implémentation des différents aspects de spécification de sécurité.
- Le cours insistera surtout sur les aspects de sécurité communs à tous les langages de programmation.
- L'exercice se devra aussi de renforcer les apprentissages techniques transversaux de la formation à savoir : la logique de programmation, la conception et la programmation objet, le langage Java.

Sur base du vécu professionnel de mes anciens étudiants, nous avons choisi 4 aspects techniques que les futurs programmeurs ont le plus de probabilité de rencontrer :

- La signature électronique du code afin de garantir la provenance du code mis en oeuvre dans les entreprises.
- La mise en place de connexions sécurisées pour garantir la confidentialité des données échangées au travers du réseau public.
- L'authentification des utilisateurs ayant accès aux ressources électroniques des entreprises.
- La gestion du contrôle d'accès à ces ressources en fonction de l'utilisation mais aussi la garantie de l'origine du code utilisé pour y avoir accès.

Pour atteindre ces objectifs, nous présenterons, dans la première partie de ce mémoire, les bases théoriques que les étudiants doivent acquérir.

Ces bases concernent :

- La compréhension des principes sur lesquels se fonde la mise en place d'un système de gestion sécurisée. Ceci envisage les principes de base de la cryptographie et l'utilisation de ceux-ci dans l'authentification par certificat et dans l'établissement d'une connexion sécurisée de type TLS/SSL.
- Les techniques spécifiquement intégrées dans la plateforme Java pour la gestion de la sécurité comme son architecture générale basée sur le principe du « bac à sable », son extension des services réseaux pour gérer les communications sécurisées JSSE(*Java Secure Socket Extension*) et enfin son service d'authentification modulaire et la liaison de celui-ci à l'architecture globale de gestion de contrôle d'accès.

La deuxième partie de ce mémoire abordera la conception méthodologique d'un exercice intégré.

Nous définirons tout d'abord les caractéristiques du public auquel est destiné cet exercice en terme de pré-requis et la réalité du groupe avec lequel ce travail va devoir être réalisé. Nous détaillerons les différentes matières du cours en objectifs opérationnels.

Nous définirons la progression de présentation des concepts et de leurs articulations avec les expérimentations concrètes afin développer progressivement la compréhension de la problématique et des solutions concrètes envisageables.

Ensuite, toujours sur base des objectifs, nous envisagerons une application concrète permettant d'expérimenter les différents aspects de sécurité. Cette application devra être suffisamment riche pour pouvoir implémenter les différentes solutions et suffisamment simple pour que la compréhension du contexte de base (l'application sans spécification de sécurité) ne prenne pas le pas sur la compréhension des aspect de sécurité.

La troisième partie présentera les documents remis aux étudiants.

Ces documents sont le déclencheur de la démarche de recherche et de conception de solutions informatiques intégrant la gestion de la sécurité. Ils ont été conçus pour susciter le questionnement et pour donner les clés permettant d'atteindre les objectifs du cours.

Le premier document présentera l'application prototype que les étudiants doivent sécuriser.

Le deuxième est une liste d'attentes d'une société fictive auxquelles doivent répondrent des candidats dans un processus de sélection.

Pour terminer, nous conclurons en reprenant les différents apprentissages que les étudiants acquièreront grâce à la solution tant pédagogique que technique développée. Nous dégagerons enfin des pistes de travail pour faire évoluer ce cours difficile.

Partie I Les techniques de sécurité

1. Objectifs d'un programme « sécurisé »

Un programme sécurisé doit être [Scott Oaks, 1999] :

- **A l'abri de programmes malfaisants** : pas de nuisances pour l'environnement hôte.
- **Non-intrusif** : pas d'accès aux données résidant sur l'hôte.
- **Authentifié** : l'identité des parties impliquées doit être vérifiée.
- **Chiffré** : toute donnée émise ou reçue par le programme doit être chiffrée.
- **Surveillé** : toute opération sensible doit être consignée.
- **Correctement défini** : des spécifications de sécurité claires doivent être rigoureusement suivies.
- **Vérifié** : les modes opératoires doivent être définis et vérifiés.
- **Economique** : le programme ne doit pas être autorisé à consommer trop de ressources systèmes.
- **Certifié** (ex: C1 ou B1 du gouvernement américain) : afin de garantir l'inclusion de certaines procédures de sécurité.

Analyse de l'intégration de ces concepts au sein de l'environnement de développement Java :

- Actuellement, seul les 4 premiers objectifs font l'objet d'une implémentation au niveau des distributions d'exécution (JRE) et de développement (SDK).
- La mise en place et la configuration des paramètres de sécurité sont de la responsabilité du système hôte (JRE local, Browser, Tomcat, ...) et ne sont pas forcément actifs par défaut.
- Pour le chiffrement des données, Java offre les outils nécessaires pour atteindre l'objectif, mais son implémentation est de la responsabilité des programmeurs (le chiffrement peut être pris en charge par le système hôte).
- Le modèle envisagé ici est celui implémenté par Sun au niveau de son SDK 1.4. Certaines distributions peuvent avoir modifié ce modèle.

2. Cryptographie et sécurisation de communication

2.1. Cryptographie

La **cryptologie** est la science qui étudie les aspects scientifiques de ces techniques, c'est-à-dire qu'elle englobe la cryptographie et la cryptanalyse.

Le mot **cryptographie** regroupe l'ensemble des techniques permettant de **chiffrer** des messages, c'est-à-dire permettant de les rendre inintelligibles sans une action spécifique.

La cryptologie est essentiellement basée sur l'arithmétique. Il s'agit d'utiliser une valeur numérique associée aux lettres qui composent un message puis ensuite de faire des opérations sur ces valeurs pour :

- D'une part, les modifier de telle façon à les rendre incompréhensibles. Le résultat de cette modification (le message chiffré) est appelé **cryptogramme** (en anglais *ciphertext*) par opposition au message initial, appelé **message en clair** (en anglais *cleartext*).
- D'autre part, faire en sorte que le destinataire puisse les déchiffrer.

Le fait de coder un message de telle façon à le rendre secret s'appelle **chiffrement** (en anglais *encryption*). La méthode inverse, consistant à retrouver le message original, est appelée **déchiffrement** (en anglais *decryption*).

Le chiffrement d'un message se fait à l'aide d'une **clé de chiffrement**, le déchiffrement nécessite quant à lui une **clé de déchiffrement**.

On distingue généralement deux types de clés :

- **Les clés symétriques** : il s'agit de clés utilisées pour le chiffrement ainsi que pour le déchiffrement. On parle alors de chiffrement symétrique ou de **chiffrement à clé secrète** (en anglais *secret key cryptography*).
- **Les clés asymétriques** : il s'agit de clés utilisées dans le cas du chiffrement asymétrique, aussi appelé **chiffrement à clé publique** (en anglais *public key cryptography*). Dans ce cas, une clé différente est utilisée pour le chiffrement et pour le déchiffrement.

La **cryptanalyse** s'oppose, en quelque sorte, à la cryptographie. En effet, si déchiffrer consiste à retrouver le texte clair au moyen d'une clé, cryptanalyser c'est tenter de se passer de cette dernière.

Sans entrer dans le détail des techniques, il est important de comprendre que si l'algorithme de chiffrement est bien conçu, toutes les techniques de cryptanalyse devront passer par la technique d'essais/erreurs sur l'ensemble

des clés possibles (attaque par force brute). Plus la taille d'une clé est grande plus le nombre de possibilités est important et donc plus faible est la probabilité de découverte de la clé dans un délai utile. À partir d'une certaine taille de clé, cette probabilité devient pratiquement nulle.

Le nombre possible de clés, pour une taille donnée, n'est pas la même pour tous les algorithmes de chiffrement. Dans un document publié fin 2006 le secrétariat général de la défense nationale française [Direction centrale de la sécurité des systèmes d'information, 2006] recommandait, pour une utilisation standard des clés symétriques, une taille de 128 bits et de 2048 bits pour la taille des clés asymétriques.

2.1.1. La cryptographie à clé secrète

Dans la **cryptographie à clé secrète**, la clé servant à chiffrer les données peut être facilement déterminée si l'on connaît la clé servant à déchiffrer. Il est aussi facile de trouver la clé de déchiffrement en connaissant la clé de chiffrement. Dans la plupart des systèmes symétriques, la clé de chiffrement est la même que la clé de déchiffrement.

La **cryptographie symétrique** est très utilisée et se caractérise par une grande rapidité.

Le principal inconvénient de ce système provient de l'échange des clés. En effet, le chiffrement symétrique repose sur l'échange d'un secret (les clés). Ainsi se pose le problème de la distribution des clés : pour un groupe de n personnes utilisant un chiffrement à clés secrètes, il est nécessaire de distribuer $n \times (n-1) / 2$ clés.

De plus l'idéal serait de changer très régulièrement de clés.

2.1.2. La cryptographie à clé publique

Dans la **cryptographie à clé publique**, les clés existent par paires:

- Une **clé publique** pour le chiffrement.
- Une **clé privée** pour le déchiffrement.

Ainsi, dans un système de chiffrement à clé publique, les utilisateurs génèrent deux clés liées entre elles. Ce qui a été chiffré par l'une ne peut être déchiffré que par l'autre. Ils choisissent de rendre publique une des deux clés (il s'agit de la clé publique) et de garder l'autre secrète (il s'agit de la clé privée). Les utilisateurs s'échangent la clé publique au travers d'un canal non sécurisé.

Lorsqu'un utilisateur désire envoyer un message à un autre utilisateur, il lui suffit de chiffrer le message à envoyer au moyen de la clé publique du destinataire. Ce dernier sera en mesure de déchiffrer le message à l'aide de sa clé privée (qu'il est seul à connaître).

Pour faire une image avec le "monde réel", il s'agit pour un utilisateur de créer une clé aléatoire (clé privée), puis de fabriquer un grand nombre de cadenas (clé publique) qu'il dispose dans un casier accessible par tous (le casier joue le rôle de canal non sécurisé). Pour lui faire parvenir un document, chaque utilisateur peut prendre un cadenas (ouvert), fermer une valisette contenant le document grâce à ce cadenas, puis envoyer la valisette au propriétaire de la clé publique (le cadenas). Seul le propriétaire sera alors en mesure d'ouvrir la valisette avec sa clé privée.

Le processus inverse consistant à chiffrer le message avec sa clé privée que chacun peut déchiffrer avec la clé publique correspondante, ne garantit évidemment pas la confidentialité mais garantit l'origine du message, car seule la personne à qui correspond la clé publique de déchiffrement a pu chiffrer le message de cette manière (avec sa clé privée).

Le problème consistant à se communiquer la clé publique n'existe plus, les clés publiques pouvant être envoyées librement. Le chiffrement par clé publique permet donc à des personnes de s'échanger des messages cryptés sans pour autant posséder de secret en commun. En contrepartie tout le challenge consiste à (s')assurer que la clé publique que l'on récupère est bien celle de la personne avec qui l'on souhaite communiquer.

2.1.3. Les clés de session

Les algorithmes asymétriques permettent de s'affranchir de problèmes liés à l'échange de clés via un canal sécurisé. Toutefois, ces derniers restent beaucoup moins efficaces (en terme de temps de calcul) que les algorithmes symétriques.

Ainsi, la notion de **clé de session** est un compromis entre le chiffrement symétrique et asymétrique permettant de combiner les deux techniques.

Le principe de la clé de session est simple : il consiste à générer aléatoirement une clé de session de taille raisonnable, et de chiffrer celle-ci à l'aide d'un algorithme de chiffrement à clé publique (plus exactement à l'aide de la clé publique du destinataire).

Le destinataire est en mesure de déchiffrer la clé de session à l'aide de sa clé privée. Ainsi, expéditeur et destinataire sont en possession d'une clé commune dont ils sont seuls connaisseurs. Il leur est alors possible de s'envoyer des documents chiffrés à l'aide d'un algorithme de chiffrement symétrique.

La clé de session peut également être négociée entre les deux parties en utilisant un processus de négociation (appelé en anglais *key agreement*) dont le plus utilisé est l'algorithme de Diffie-Hellman (voir annexe A.I.) du nom de ses inventeurs Diffie et Hellman [E.Rescorla, 1999]. Dans cette technique, les deux parties génèrent des informations privées qui combinées entre elles donneront la clé de session. Toute personne récupérant les informations entre les deux parties sera incapable de reconstituer la clé sans connaître au moins une des deux informations secrètes.

2.1.4. Hachage

Une **fonction de hachage** (parfois appelée fonction de condensation) est une fonction permettant d'obtenir le condensé d'un texte, c'est-à-dire une suite de caractères assez courte représentant le texte qu'il condense.

Ainsi, une fonction de hachage doit remplir quelques conditions de base :

- Le texte d'origine en entrée peut être de dimension variable. Le condensé en sortie doit être fixe.
- Elle doit être relativement facile à calculer.
- Elle doit être une fonction à sens unique. Il ne doit pas être possible de retrouver le message original à partir du résultat de la fonction.
- Elle doit être "sans collision". Toute modification dans le message original doit provoquer une modification au niveau du condensé.

Les algorithmes de hachage les plus utilisés actuellement sont :

- **MD5** (MD signifiant *Message Digest*), créant une empreinte digitale de 128 bits [R.Rivest, 1992].
- **SHA** (pour *Secure Hash Algorithm*, pouvant être traduit par *Algorithme de hachage sécurisé*) créant des empreintes d'une longueur de 160 bits

2.1.5. Code d'authentification de message

Un code d'authentication de message (en anglais *Message Authentication Code,* MAC) permet de vérifier l'intégrité de l'information transmise à travers un support non sécurisé.

Le mécanisme utilisé est basé sur une fonction de hachage telle que MD5 ou SHA appliquée sur le texte combiné avec une clé secrète. Ce mécanisme appelé *Keyed-Hashing for Message Authentication (HMAC)* est spécifié par la RFC 2104 [H. Krawczyk IBM, M. Bellare UCSD, R. Canetti IBM, 1997].

2.1.6. Les fonctions de la cryptographie

La cryptographie est traditionnellement utilisée pour dissimuler des messages aux yeux de certains utilisateurs. Cette utilisation a aujourd'hui un intérêt d'autant plus grand que les communications via Internet circulent dans des infrastructures dont on ne peut garantir la fiabilité et la confidentialité. Désormais, la cryptographie sert non seulement à préserver la confidentialité des données mais aussi à garantir leur intégrité et leur authenticité.

- La **confidentialité** consiste à rendre l'information inintelligible à d'autres personnes que les acteurs de la transaction.
- L'**authentification** consiste à assurer l'identité d'un utilisateur, c'est-à-dire de garantir à chacun des correspondants que son partenaire est bien celui qu'il croit être. Un contrôle d'accès peut permettre (par exemple au moyen

d'un mot de passe qui devra être crypté ou d'un certificat électronique) l'accès à des ressources uniquement aux personnes autorisées.

Grâce à la signature électronique, il est possible :

- De vérifier l'**intégrité** des données c'est-à-dire de déterminer si les données n'ont pas été altérées durant la communication (de manière fortuite ou intentionnelle).
- De garantir la **non-répudiation** de l'information, c'est-à-dire garantir qu'aucun des correspondants ne pourra nier la transaction.

2.2. Signature et authentification électroniques

2.2.1. Signature électronique

Les **signatures numériques** sont fondamentales au niveau de l'authentification, de l'identification d'entité, de l'autorisation et de la non-répudiation. Le but est de fournir des moyens à une entité de pouvoir lier son identité à une information.

Une personne voulant assurer le destinataire qu'elle est bel et bien la responsable du message, chiffrera ce message avec sa clé privée et le destinataire déchiffrera le message chiffré avec la clé publique correspondante de l'expéditeur.

Habituellement, une fonction de hachage est utilisée pour créer une empreinte du message et le chiffrement à l'aide de la clé privée est appliqué sur l'empreinte uniquement.

Voici comment fonctionne une communication sécurisée avec signature électronique :

1 L'expéditeur calcule l'empreinte de son message à l'aide d'une fonction de hachage.

2 L'expéditeur chiffre l'empreinte avec sa clé privée.

3 L'expéditeur réalise l'empreinte chiffrée au départ du texte clair à l'aide de la clé publique du destinataire ou de la clé de session.

4 L'expéditeur envoie le message chiffré au destinataire.

5 Le destinataire déchiffre le message avec sa clé privée ou la clé de session.

6 Le destinataire déchiffre l'empreinte avec la clé publique de l'expéditeur.

7 Le destinataire calcule l'empreinte du texte clair à l'aide de la même fonction de hachage que l'expéditeur.

8 Le destinataire compare les deux empreintes.

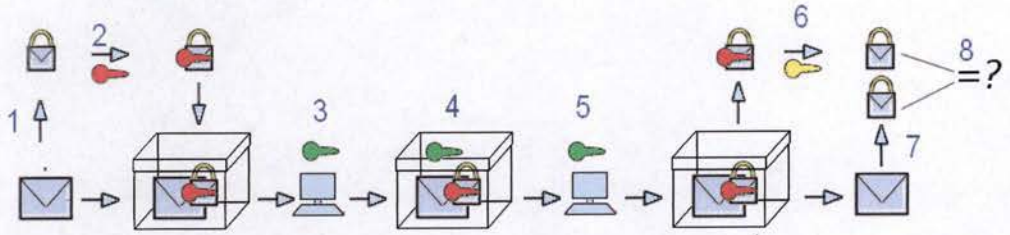

Figure 1 : Fonctionnement d'une communication sécurisée avec signature électronique

2.2.2. Certificat électronique.

Les algorithmes de chiffrement asymétrique sont basés sur le partage entre les différents utilisateurs d'une clé publique. Ce mode de partage a une grande lacune : **rien ne garantit l'identité de l'utilisateur à qui elle est associée**. En effet, un pirate peut distribuer sa clé publique sous une autre identité. Ainsi, le pirate sera en mesure de déchiffrer tous les messages ayant été chiffrés avec la clé associée à la fausse identité.

Un certificat permet d'associer une clé publique à une entité (une personne, une machine, ...) afin d'en assurer la validité. Le certificat est une sorte de carte d'identité avec une clé publique, délivré par un organisme appelé autorité de certification (**CA** pour *Certification Authority*).

L'autorité de certification est chargée de délivrer les certificats, de leur assigner une date de validité, ainsi que de révoquer éventuellement des certificats avant cette date en cas de compromission de la clé (ou du propriétaire).

Qu'est-ce qu'un certificat ?

Les certificats sont des petits fichiers divisés en deux parties :

- La partie contenant les informations.
- La partie contenant la signature de l'autorité de certification.

La structure des certificats est normalisée par le standard **X.509** de l'UIT (Union Internationale des Télécommunications) qui définit les informations contenues dans le certificat :

- Le nom de l'autorité de certification.
- Le nom du propriétaire du certificat.
- La date de validité du certificat.
- L'algorithme de chiffrement utilisé.
- La clé publique du propriétaire.

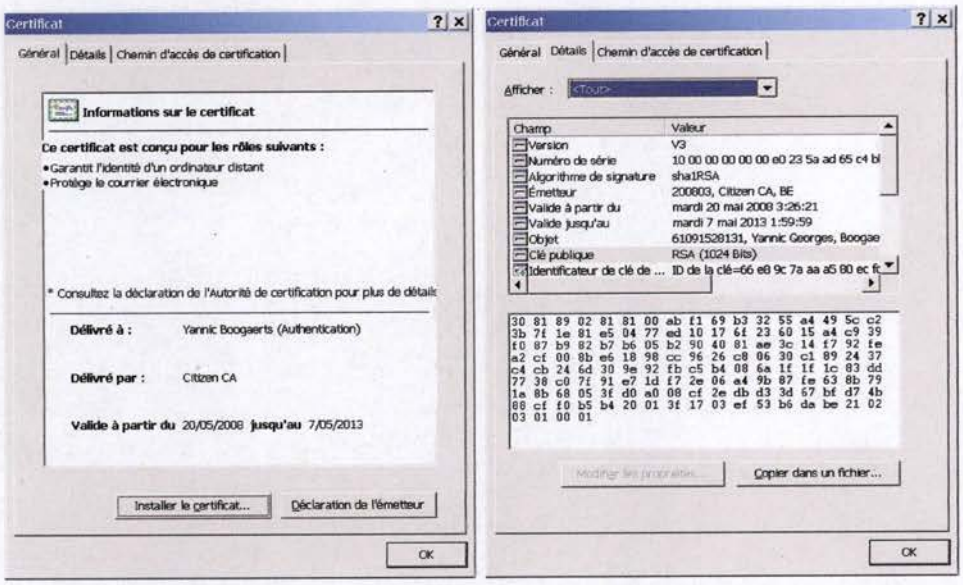

Figure 2 : Certificat contenu sur la puce d'une carte d'identité

L'ensemble de ces informations (informations et clé publique du demandeur) est signé par l'autorité de certification. Cela signifie qu'une fonction de hachage crée une empreinte de ces informations, puis ce condensé est chiffré à l'aide de la clé privée de l'autorité de certification. La clé publique de l'autorité de certification permet aux utilisateurs de vérifier la signature avec la clé publique.

2.3. Protocole de communication sécurisé TLS

Transport Layer Security (**TLS**), construit à patir de ***Secure Socket Layer*** (**SSL**), est un protocole de sécurisation des échanges sur Internet spécifié actuellement par la RFC 5246 [T. Dierks Independent, E. Rescorla RTFM, Inc, 2008]. Il y a très peu de différences entre SSL version 3 et TLS version 1.

TLS fonctionne suivant un mode client-serveur. Il fournit les objectifs de sécurité suivants :

- L'authentification du serveur,.
- La confidentialité des données échangées.
- L'intégrité des données échangées.
- De manière optionnelle, l'authentification du client avec l'utilisation d'un certificat numérique.
- La spontanéité, à savoir qu'un client peut se connecter pour la première fois à un serveur de façon transparente.
- La transparence, qui a contribué certainement à sa popularité, du fait que les protocoles de la couche application n'aient pas à être modifiés pour utiliser une connexion sécurisée par TLS. Par exemple, le protocole HTTP est identique, que l'on se connecte à un schème HTTP ou HTTPS.

TLS/SSL est un protocole se situant entre la couche transport et la couche application de la pile TCP/IP.

TCP/IP Layer	Protocol
Application Layer	HTTP, NNTP, Telnet, FTP, etc.
Secure Sockets Layer	**SSL**
Transport Layer	TCP
Internet Layer	IP

Figure 3 : Pile des protocoles TCP/IP avec SSL

TLS/SSL met en place une session chiffrée entre le client et le serveur. Les techniques de chiffrement sont multiples :

- Un chiffrement asymétrique comme par exemple l'algorithme **RSA** pour l'authentification.
- Un chiffrement symétrique (plus léger à réaliser qu'un chiffrement asymétrique) comme par exemple le **AES** pour assurer la transmission confidentielle des informations.
- Une fonction de hachage, comme le **SHA-1**, pour s'assurer que les données soient transmises sans être corrompues.

Le choix des algorithmes et la longueur des clés utilisées seront négociés à l'établissement de la session entre le client et le serveur.

2.3.1. Processus TLS/SSL

La communication TLS/SSL commence par un échange d'informations entre le client et le serveur appelé prise de contact *handshake*. L'objectif de cette première phase est :

- La négociation de la suite de chiffrement (en anglais *cipher suite*).
- L'authentification des parties (optionnel).
- L'initialisation des propriétés de sécurité utilisées pour le chiffrement des données.

Négociation de la suite de chiffrement

Une suite de chiffrement est un ensemble d'algorithmes et de tailles de clé de chiffrement. Elle reprend :

- L'algorithme d'échange de clé publique ou l'algorithme de détermination de clé.

- L'algorithme de chiffrement symétrique pour la confidentialité des données.
- La fonction de hachage.

Le client envoie au serveur l'ensemble des suites de chiffrement qu'il est capable de supporter et le serveur choisit la meilleure suite acceptable par les deux parties.

Authentification des parties

L'authentification est optionnelle, mais il est fréquent que le client doive avoir une garantie de l'identité du serveur avant, par exemple, d'envoyer des informations confidentielles sur le canal chiffré.

Pour prouver que le serveur appartient bien à l'organisation qu'il représente, le serveur envoie un certificat avec sa clé publique. En validant ce certificat grâce à la clé publique de l'autorité de certification, il a la garantie de communiquer avec le bon serveur.

Il est possible aussi de configurer l'authentification du client par le serveur en utilisant la même technique. Cependant, cette option est plus rarement utilisée car peu de clients ont un certificat les authentifiant ou ne sont pas prêts à l'utiliser.

Par la suite, le client et le serveur échangent des informations pour décider d'une clé secrète symétrique pour la session. Par exemple, avec RSA le client utilise la clé publique du serveur contenue dans son certificat pour chiffrer la clé et l'envoie au serveur. Comme seul le serveur peut déchiffrer cette information, la clé ne sera connue que par le client et le serveur.

Envoi chiffrées de donnée

Le client et le serveur utilisent la clé symétrique avec l'algorithme choisi au moment de la négociation pour chiffrer les données et fournir un HMAC de celle-ci.

2.3.2. Description du protocole TLS/SSL

Le chapitre précèdent donnait une description générale du protocole. Nous allons ici décrire plus en détail les messages échangés entre le client et le serveur.

La figure ci-dessous présente la séquence des messages échangés pendant la phase de prise de contact. Les messages qui ne sont envoyés que dans certaines circonstances sont notés comme optionnels.

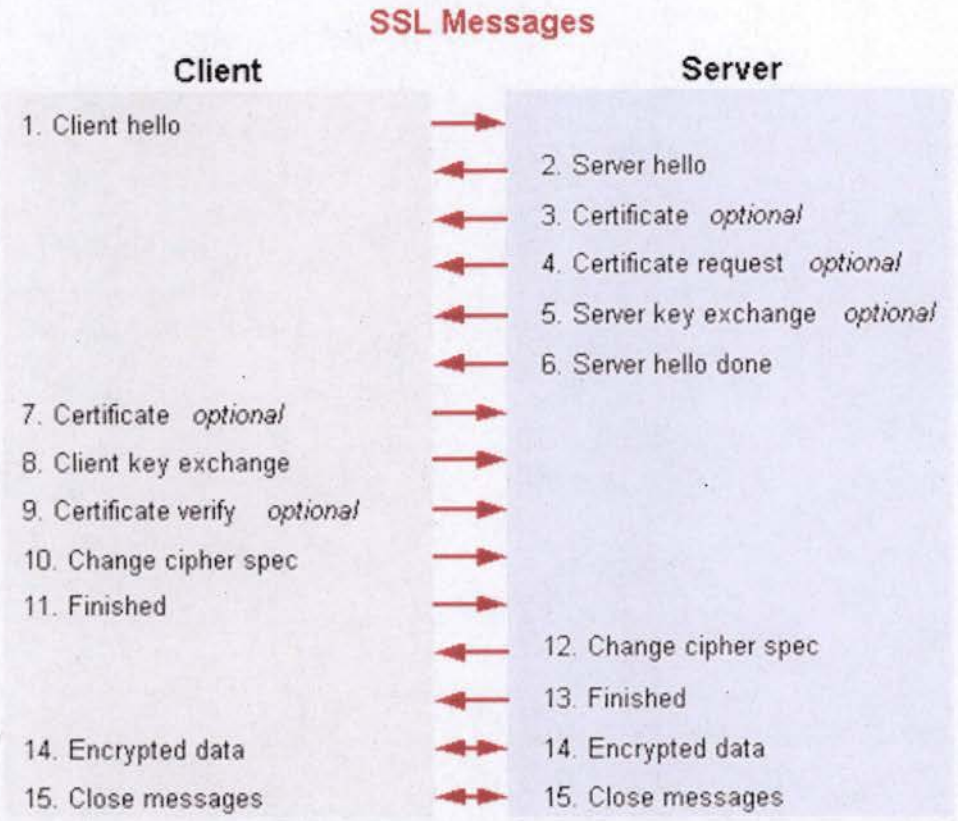

Figure 4 : Séquence des messages échangés dans le protocole TLS/SSL

1 ***Client hello*** **:** le client envoie au serveur la version la plus haute du protocole qu'il est capable de supporter et la liste des suites de chiffrement qu'il supporte.

2 ***Server hello*** **:** le serveur choisit la version la plus haute du protocole et la meilleure suite de chiffrement qu'ils supportent tous les deux et les envoie au client.

3 ***Certificate*** **:** le serveur doit envoyer un message *Certificate* chaque fois que la méthode d'échange de clés acceptée (contenue dans la suite de chiffrement) utilise des certificats pour l'authentification, c'est-à-dire toutes les méthodes d'échange de clés sauf les échanges Diffie-Hellman anonymes (DH_anon). Le serveur envoie au client un certificat ou une chaîne de certificats. Une chaîne de certificats commence par le certificat à clé publique du serveur et se termine par celui de l'autorité de certification principale. Ce message est optionnel mais est envoyé lorsque l'authentification du serveur est demandée.

4 ***Certificate request*** **:** si le serveur a besoin d'authentifier le client, il lui envoie une requête de certificat.

5 ***Server key exchange*** **:** le message *ServerKeyExchange* n'est envoyé par le serveur que lorsque le message *Certificate* du serveur (s'il est envoyé) ne contient pas assez de données pour permettre au client d'échanger un modèle initial de secret. Ceci est vrai pour les méthodes d'échange de clés utilisant Diffie-Hellman.

6 ***Server hello done*** : le message *ServerHelloDone* est envoyé par le serveur pour indiquer la fin du *ServerHello* et des messages associés. Il signifie que le serveur a fini d'envoyer les messages de prise en charge de l'échange de clés et que le client peut passer à sa phase d'échange de clés.

7 ***Certificate*** : Ce message n'est envoyé que si le serveur demande un certificat. Si aucun certificat convenable n'est disponible, le client envoie un message *Certificate* ne contenant aucun certificat. Si le client n'envoie aucun certificat, le serveur peut à sa discrétion, soit continuer la prise de contact sans authentification du client, soit répondre par une erreur et mettre fin au processus.

8 ***Client key exchange*** : avec ce message, la clé secrète est établie, soit par transmission directe de la clé chiffrée en RSA, soit par la transmission des paramètres Diffie-Hellman qui vont permettre à chaque partie de s'accorder sur la même clé secrète.

9 ***Certificate verify*** : ce message est utilisé pour fournir une vérification explicite d'un certificat de client. Ce message n'est envoyé qu'à la suite d'un certificat de client qui a la capacité de signature. Le client chiffre tous les messages échangés jusqu'à présent avec sa clé privée. Grâce à cette valeur et à la clé publique contenue dans le certificat, le serveur pourra vérifier que le client possède bien la clé privée correspondante et par là, valider l'authentification du client.

10 ***Change cipher spec*** : le client envoie un message stipulant que les messages suivants utiliseront le type de chiffrement négocié.

11 ***Finished*** : le message *Finished* est le premier qui est protégé avec l'algorithme et la clé secrète qui viennent juste d'être négociés. Le serveur doit vérifier que le contenu est correct.

12 ***Change cipher spec*** : le serveur envoie un message stipulant que les messages suivants utiliseront le type de chiffrement négocié.

13 ***Finished*** : le serveur envoie lui aussi un message *Finished* chiffré tel que négocié.

14 ***Encrypted data*** : une fois qu'une des parties a envoyé son message *Finished* et reçu et validé le message *Finished* de son homologue, il peut commencer à envoyer et recevoir des données d'application via la connexion. Les données seront communiquées en utilisant l'algorithme de chiffrement et la fonction de hachage négociés avec les messages 1 et 2 et la clé secrète finalisée avec le message 8.

15 ***Close Messages*** : à la fin de la connexion les deux parties envoient un message *Close_notify* pour informer son partenaire que la connexion est fermée.

2.3.3. Impact du choix de la suite de chiffrement

Le choix de la suite de chiffrement a un impact direct sur la sécurité de la communication. Par exemple, si la suite de chiffrement choisie précise une

connexion anonyme, l'application ne peut pas garantir l'identité du partenaire avec lequel elle communique de manière confidentielle. De même, la suite de chiffrement négocié peut spécifier qu'aucun algorithme de chiffrement ne doit être utilisé. Dans ce cas, la confidentialité des données n'est plus garantie.

Attention : le protocole n'a aucun mécanisme pour vérifier que l'identité du serveur corresponde à la requête initiale qui a été envoyée. La seule vérification est que son certificat soit valide et qu'il soit signé par une autorité digne de confiance. Il est donc primordial que l'application qui utilise TLS/SSL mette en place un mécanisme de vérification de cette correspondance.

3. Modèle de sécurité Java

3.1. Spécification du langage contribuant à la sécurité

Protection de la mémoire vive : une application ne peut accéder qu'à ses propres données et uniquement par l'intermédiaire des classes qui en réglementent l'accès.

Voici les règles d'accessibilité :

- Les variables de type primitif ne peuvent être accédées qu'à l'aide de la classe ou de l'instance qui les contient.
- Les tableaux et les instances sont accessibles à partir de leur adresse contenue dans des variables de type référence. Aucune opération autre que l'assignation, l'accès à un membre, ou le transtypage vers un type descendant n'est permise. Il n'y a donc pas d'équivalent en Java aux pointeurs de façon à garantir un accès contrôlé aux propriétés de la classe.
- Les références ne peuvent être assignées à une variable de type primitif ni être transtypées vers un de ces types et inversement.
- Les limites d'un tableau sont strictement contrôlées.
- Une variable de type référence ne peut être assignée que par la référence à une instance de même type ou d'un de ses descendants.
- Toute variable doit être initialisée avant d'être lue.

Le langage prévoit différentes règles permettant de limiter l'accès aux classes et instances ainsi qu'à leurs propriétés et méthodes :

- Tout élément déclaré comme ***public*** peut être accédé librement.
- Tout élément qui ne fait pas l'objet d'une déclaration spécifique est considéré comme ***privé au package*** et ne peut être accédé qu'à partir des méthodes d'une classe du même *package*.
- Les méthodes, propriétés et classes internes déclarées comme ***protected*** ne sont accessibles qu'aux méthodes d'une classe du même *package* ou d'une classe descendante.
- Les méthodes, propriétés et classes internes déclarées comme ***private*** ne sont accessibles qu'aux méthodes de la classe où elles sont implémentées.

Attention : lorsqu'une instance est sérialisée, l'accès à ses propriétés n'est plus protégé.

Les classes ou méthodes déclarées comme ***final*** ne peuvent pas être surchargées.

3.2. Architecture de sécurité de Java

3.2.1. Architecture d'une application Java [Li Gong, 2002]

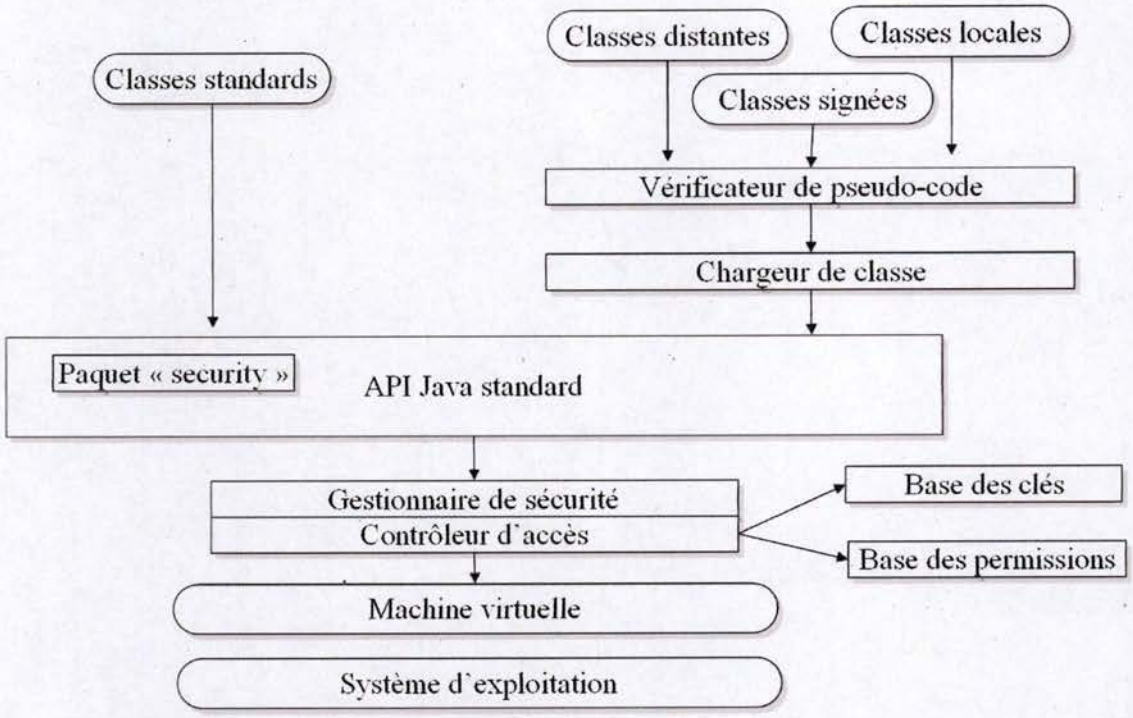

Figure 5 : Architecture des composants participant à la sécurité

Vérificateur de pseudo-code

Il s'assure que le contenu des fichiers Java (*bytecode*) est conforme aux règles du langage.

Cette vérification n'ajoute rien à ce que le compilateur « javac.exe » vérifie déjà, mais il est possible que le compilateur utilisé ne soit pas conforme ou que l'utilisation de deux fichiers « class » (fichiers contenant le code compilé en *bytecode*), compilés à des moments différents, contiennent des accès illégaux.

Par exemple, dans une version 1, une classe A possède le code qui accède à une propriété de portée publique d'une classe B. Puis, dans une version 2, la propriété de B est devenue d'accès privé. Si l'on combine le *bytecode* de la classe A de la version 1 avec celui de la classe B de la version 2, nous avons deux codes compilés avec un compilateur conforme qui exécute un code non conforme.

Toutes les opérations du programme ne sont pas vérifiées au moment du chargement de la classe. Par exemple, la vérification de l'accès aux champs d'une classe n'est effectuée qu'au moment du premier appel de cet accès.

Chargeur de classe

Les classes sont chargées en mémoire via différents chargeurs de classes chaînés entre eux et dont le dernier est le chargeur système dédié aux classes de l'API.

Un chargeur de classes ne chargera la classe que si :

- L'application a le droit d'utiliser cette classe.
- Le chargeur suivant dans la chaîne n'a pas réussi à la charger.
- La classe n'existe pas déjà en mémoire ce qui garantit, par exemple, qu'une classe de l'API Java gérant la sécurité ne peut pas être remplacée par une autre venant de l'extérieur.

Chaque chargeur de classe est dédié à une ou plusieurs localisations.

Cette localisation ainsi que les signataires sont ajoutés aux propriétés de la classe au moment du chargement (code source).

Deux classes de même nom de *package* mais chargées par deux chargeurs de classes différentes ne seront pas considérées comme faisant partie du même *package* et ne pourront donc pas accéder aux membres de portée ***package*** de l'autre classe.

Gestionnaire de sécurité

Java propose une classe représentant un point central de la gestion de sécurité (`java.lang.SecurityManager`). C'est l'instance active de cette classe qui est en charge de la vérification des permissions.

Seule la référence de l'instance renvoyée par la méthode `System.getSecurityManager()` peut être considérée comme la référence du gestionnaire de sécurité actif. Si la méthode retourne une valeur nulle, c'est qu'aucune politique de sécurité n'est appliquée pour le moment.

La sélection du gestionnaire de sécurité référencée par le système peut se faire de deux manières différentes :

- Au moment du lancement de la machine virtuelle, par l'ajout d'une de ces versions du paramètre :
  - `-Djava.security.manager` (dans ce cas le *SecurityManager* par défaut est utilisé).
  - `-Djava.security.manager=<className>` (dans ce cas la classe dont le nom remplace `<className>` est utilisée comme gestionnaire de sécurité).
- Par programmation, la méthode `System.setSecurityManager()` permet de modifier le gestionnaire de sécurité actif. Si le système référence un *SecurityManager* au moment de l'appel de la méthode, un appel à `checkPermission()` avec comme paramètre une

`RuntimePermission("setSecurityManager")` sera exécuté afin de vérifier si cette action est autorisée.

Le gestionnaire de sécurité par défaut proposé par l'API Java utilise les méthodes du contrôleur d'accès pour la vérification des permissions.

L'application systématique du contrôle d'accès aux ressources protégées par le « bac à sable » est garantie par le fait que :

- Toutes les méthodes de l'API accédant à des fonctionnalités critiques utilisent cette procédure.
- Toute application est obligée d'utiliser les méthodes de l'API pour accéder aux ressources.

Dans l'implémentation de Java proposée par Sun, le *SecurityManager* délègue la vérification des permissions au contrôleur d'accès (*AccessController*). Le gestionnaire de sécurité est en charge de la vérification des permissions et le contrôleur d'accès implémente une manière d'appliquer cette vérification.

Contrôleur d'accès

Le rôle du contrôleur d'accès est le même que celui du gestionnaire de sécurité. Les différences résident dans son implémentation :

- Le contrôleur d'accès est représenté par une classe unique `java.security.AccessController`. Il ne peut exister d'instance de la classe car elle ne peut être instanciée (son constructeur est privé). Par contre, cette classe comprend un certain nombre de méthodes statiques appelées pour déterminer si une opération donnée doit aboutir. La méthode clé de cette classe prend en argument une permission et détermine si cette permission doit être accordée en se basant sur la politique de sécurité indiquée par l'instance active de `Policy`.
- `public static void checkPermission(Permission p)` vérifie si la permission passée en argument doit être accordée dans le cadre de la politique de sécurité. Si cette permission est accordée, la méthode rend la main; dans le cas contraire, une exception `java.security.AccessControlException` est soulevée.

Le Contrôleur d'accès autorise ou refuse une opération donnée suivant les domaines de sécurité appartenant au contexte d'exécution.

Les spécifications du contrôleur d'accès reposent sur 3 concepts que nous détaillerons par la suite :

- **Code source :** encapsulation de l'origine des classes Java.
- **Permission :** encapsulation des requêtes de permission.
- **Domaines de protection :** encapsulation d'un code source et des permissions accordées en fonction de celui-ci.

Paquets Security

Les *packages* `java.security.*` complétés par les *packages* `javax.security.*` contiennent les classes de base permettant d'implémenter des applications sécurisées.

`java.security.*` reprend :

- Les classes permettant la gestion et la création des permissions.
- Les classes permettant la gestion des clés de chiffrement.
- Les classes permettant la gestion des signatures.
- Les classes permettant la gestion des certificats.

`javax.security.*` contient les classes permettant la gestion de l'authentification des utilisateurs.

Base de données de sécurité

La distribution de base de Java propose des implémentations rudimentaires pour :

- Le stockage des permissions.
- La gestion des certificats et clés privées.

Pour la base de données des permissions, Sun propose une classe surchargeant `java.security.Policy` permettant de récupérer, dans un fichier texte, la définition des permissions accordées et de leurs domaines de protection. Il propose également un outil (« policytool ») pour gérer et générer les fichiers de permissions.

Pour la base de données des clés, Sun propose un utilitaire « keytool » pour gérer en ligne de commande une base de données pouvant contenir et générer des clés et des certificats. Cette base de données peut également être gérée à partir d'un programme Java.

L'architecture de l'API Java permet de "facilement" créer des systèmes plus performants ou d'utiliser des applications tierces dédiées à ces objectifs.

3.2.2. Concept de « bac à sable » en Java

Le modèle de sécurité proposé par la plateforme Java est appelé « bac à sable » (*sandbox model*). Il est construit autour d'un postulat de base déterminant que la confiance que l'ont peut accorder à un code dépend de la connaissance que l'on a de son origine. Les permissions accordées à une application pour accéder aux ressources sensibles d'un système seront limitées en fonction de l'origine du code.

L'origine d'une classe est déterminée au moment de son chargement. Le chargeur de classe associe à la classe, son origine sous la forme d'une URL. Le

chargeur de classe valide également les signatures du code et associe à la classe les certificats des signataires.

Le concept « bac à sable », qui au départ n'est basé que sur l'origine du code, peut être étendu à l'utilisateur authentifié. Les permissions dans ce cas ne seront plus uniquement accordées en fonction de l'origine du code mais aussi en fonction d'identifiants d'utilisateurs.

L'ensemble des informations sur l'origine d'un classe et les permissions accordées en fonction de ces informations définissent un domaine de protection.

3.2.3. Domaine de protection

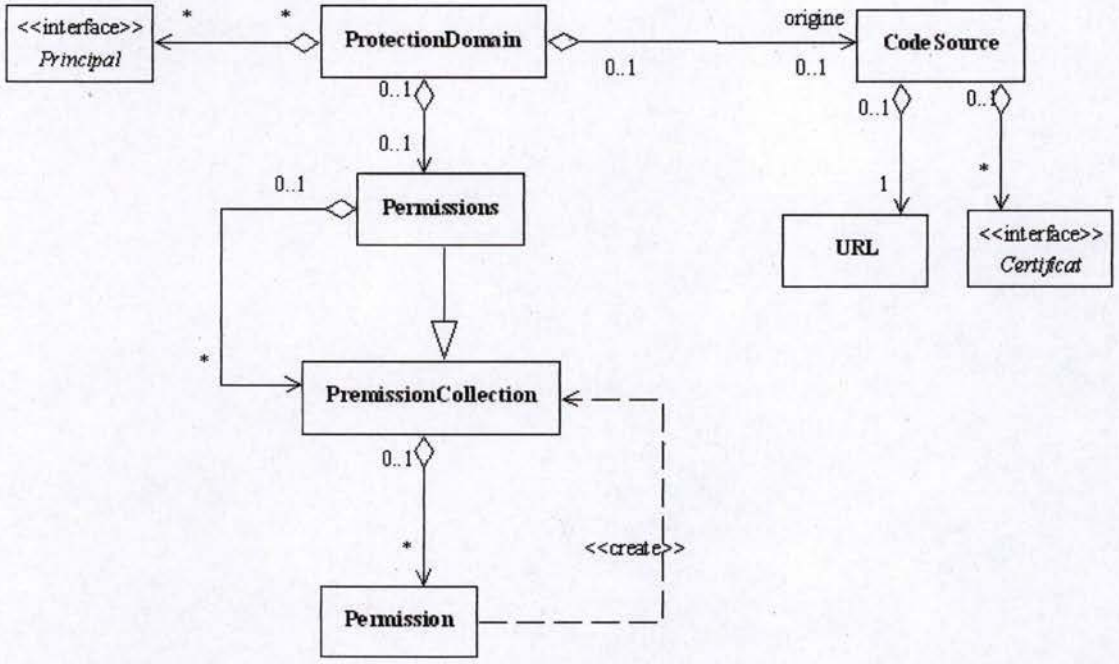

Figure 6 : Diagramme de classes représentant les principales relations entre les classes composant un domaine de protection

Un `java.security.ProtectionDomain` est défini par :

- Une **origine** du type `CodeSource` représentant l'origine du domaine de protection.
- Une **collection de permissions** du type `Permissions` représentant les permissions accordées au domaine en fonction de son origine.
- Une **collection d'identifiants** du type `Principal[]` représentant les identifiants de l'utilisateur reconnu par le système.

Code source

Un code source représente une origine. Quand il est associé à une classe à travers son domaine de protection, il représente l'origine du code de la classe.

La classe `java.security.CodeSource` est définie par :

- Une **URL** de type `URL` représentant la localisation du code source.
- Une **collection de certificats** de type `Certificate[]` authentifiant les signataires du code.

Permissions

En java les permissions sont représentées par des instances de classes étendant la classe abstraite `java.security.permission`.

Une permission, représente l'accès à une ressource. Par exemple, la permission suivante représente l'accès en lecture au fichier « formation.txt » situé dans le répertoire « C:/tmp » :

```
perm = new java.io.FilePermission("C:/tmp/formation.txt",
"read");
```

Ces accès sont définis par deux propriétés communes à toutes les permissions :

- Un **nom** de type `String` représentant souvent la ressource ou les ressources sur lesquelles portent la permission.
- Une **action** de type `String` représentant souvent l'action ou les actions sur la ressource.

Le type des classes étendant `Permission` est également utilisé par l'API dans le processus de gestion des permissions. Deux permissions de type différent ne seront jamais, par exemple, confrontées pour savoir si l'une implique l'autre.

L'imprécision au niveau de la définition des propriétés tient au fait que suivant le type de permission les notions de ressource et d'action recouvrent des réalités très différentes. Par exemple, le fait de pouvoir ou non instancier un `ClassLoader` ne porte pas sur une ressource particulière mais est simplement une fonctionnalité accordée ou non par le système hôte.

La méthode primordiale de toutes les permissions est la méthode `implies(Permission perm)`. Cette méthode crée une relation d'implication entre les permissions.

Définition : A implique B si et seulement si accorder un accès A implique que l'accès B est accordé automatiquement.

La relation d'implication entre les permissions prend une part importante dans le processus de contrôle d'accès.

Collection de permissions

Une collection de permissions représente un ensemble de permissions de même type collaborant entre elles pour déterminer si cette collection implique une permission. La contrainte que toutes les permissions soient de même type n'est

pas implémentée par l'API et est même transgressée dans le cas de la classe `java.security.Permissions` (nous y reviendrons).

Définition : la collection A implique la permission B si et seulement si accorder l'ensemble des accès contenus dans A implique que l'accès B est accordé automatiquement.

Il est possible qu'aucune des permissions contenues dans A n'implique la permission B mais que la combinaison des permissions de A implique la permission B.

Par exemple, la `CollectionPermission` A contient deux permissions perm1 et perm2 :

```
perm1 = new java.io.FilePermission("C:/tmp/*", "read");

perm2 = new java.io.FilePermission("C:/tmp/*", "delete");

B =new java.io.FilePermission("C:/tmp/formation.txt", "read,
delete");
```

Dans ce cas :

- Perm1 n'implique pas B.
- Perm2 n'implique pas B.
- Mais A implique B.

La collection Permissions

Bien que la classe `java.security.Permissions` surcharge la classe `java.security.CollectionPermission`, elle n'est pas au sens strict une collection de permissions. Elle est une collection de collections de permissions.

Chaque permission ajoutée à une instance de la classe est stockée dans une `CollectionPermissions` correspondant à son type.

Cette caractéristique est directement liée à la sémantique de sa relation d'implication.

Définition : la `CollectionPermission` A implique la permission B si et seulement si une de ces propositions est vérifiée :

- A contient la permission `AllPermission`.
- A contient une instance C de `CollectionPermission` regroupant les permissions du même type que B et que C implique B.

Principal

Un *Principal* est un identifiant représentant une identité d'un utilisateur authentifié. Cet identifiant peut le représenter lui-même ou représenter un groupe auquel il appartient.

Policy

Le système de politique de sécurité en Java définit quelles permissions sont accordées en fonction de différentes origines. Plus concrètement, c'est la classe étendant `java.security.Policy` désignée par le système hôte qui aura la responsabilité de fournir l'ensemble des permissions accordées en fonction d'un *CodeSource*.

C'est l'implémentation de cette classe qui définit sous quel format doivent être stockées les permissions sur le système hôte. Par exemple, comme un fichier ASCII, comme un fichier sérialisé ou comme une base de données.

Le choix de l'implémentation que le système hôte utilise est paramétrable via le fichier de configuration « java.security ».

La distribution de base de chez Sun propose une implémentation où les permissions sont configurables à partir d'un fichier plat ASCII [Sun Microsystems, Inc, 2002a]. C'est cette implémentation qui sera utilisée dans l'exercice ci-après.

3.2.4. Contrôle d'accès

Le gestionnaire de sécurité est chargé de vérifier si dans le contexte actuel une permission donnée est accordée ou non à l'application.

Toute méthode voulant donc conditionner son exécution au fait que le gestionnaire de sécurité en place le permet devra :

- Récupérer la référence du *SecurityManager* auprès de la classe *System*.
- Si elle n'est pas nulle, créer une instance de `Permission` représentant la permission demandée,
- Appeler la méthode `checkPermission(Permission)` du *SecurityManager* en lui passant la permission demandée.
- Si la permission est refusée, une *SecurityException* sera soulevée.

Pour vérifier si une permission peut ou non être accordée, le *SecurityManager* délègue la vérification au contrôleur d'accès qui doit :

- Récupérer les classes dont une méthode est active dans la pile des appels de méthodes.
- Récupérer auprès de chaque classe le domaine de protection auquel elle est liée.

- Créer le contexte d'exécution à partir des domaines de protection récupérés.
- Interroger le contexte pour vérifier si tous les domaines de protection du contexte impliquent la permission demandée.

Pratiquement :

- Le contrôleur d'accès (`java.security.AccessController`) récupère la pile d'appel auprès de la machine virtuelle.
- Puis crée, à partir des domaines de protection, un contexte de contrôle d'accès (`java.security.AccessControlContext`).
- Enfin, délègue la vérification de la permission à ce contexte.

Contexte d'exécution

Le contexte d'exécution à un moment donné est composé des domaines de protection liés aux classes des méthodes se trouvant dans la pile des appels de méthode à ce moment.

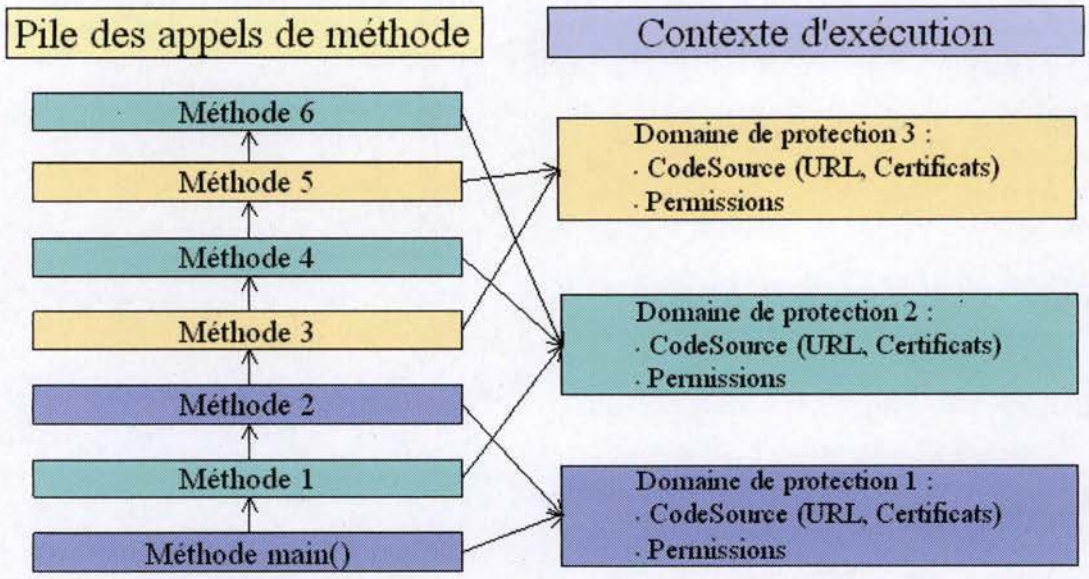

Figure 7 : Définition d'un contexte d'exécution en fonction de l'état de la pile d'exécution

Dans le scénario illustré sur le schéma ci-dessus :

- La méthode 6 est en cours d'exécution et elle a été appelée par la méthode 5 qui a été appelée par la méthode 4 et ainsi de suite jusqu'à la méthode `main()`.
- La méthode `main()` et la méthode 2 sont associées au domaine de protection 1.
- Les méthodes 1, 4 et 6 sont associées au domaine de protection 2.
- Les méthodes 3 et 5 sont associées au domaine de protection 3.
- Le contexte d'exécution est donc composé des domaines de protection 1, 2 et 3.

Il est possible d'ajouter des informations au contexte de contrôle d'accès grâce au "combineur de domaine" (*java.security.DomainCombiner*) passé en paramètre au moment de l'instanciation du contexte.

Le contexte dans ce cas devra mettre à jour les permissions en tenant compte de ces informations.

C'est cette technique qui est utilisée pour ajouter les identifiants des utilisateurs authentifiés aux domaines de protection (`Principal[]`).

Illustration : un frigo sécurisé

Pour illustrer le besoin de tenir compte de l'ensemble des domaines de protection dans l'évaluation d'une permission, ainsi que la technique à mettre en œuvre, prenons un exemple de la vie de tous les jours.

Situation

Pendant mon absence, ma fille en congé et un maçon occupé à effectuer des travaux sont présents chez moi au même moment.

Je désire réglementer l'accès au frigo en mon absence. Le maçon n'a pas la permission d'accéder au frigo tandis que ma fille bien.

Imaginons que le frigo soit en mesure de vérifier ces règles :

- Je modifie la procédure de mon frigo pour qu'il vérifie auprès du responsable de la sécurité s'il a la permission de donner une boisson.
- J'accorde la permission "donner une boisson" au domaine famille.
- Le responsable de la sécurité fait partie du domaine système qui a tous les droits.
- Ma fille et le frigo font partie du domaine famille.
- Le maçon fait partie du domaine étranger.

Scénario 1 : ma fille demande un jus d'orange au frigo

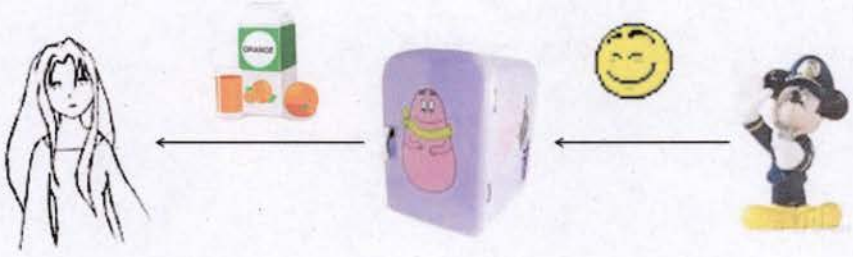

Figure 8 : Scénario 1 : ma fille demande un jus d'orange au frigo

Le contexte d'appel est composé de deux domaines de protection différents :

- Ma fille liée au domaine famille.

- Le frigo lié au domaine famille.
- Le contrôleur d'accès lié au domaine système.

Tous les domaines du contexte ont des permissions qui impliquent "donner jus d'orange", le frigo délivre le jus d'orange.

Scénario 2 : le maçon demande une bière au frigo

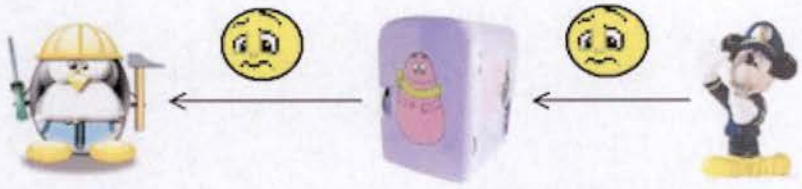

Figure 9 : Scénario 2 : le maçon demande une bière au frigo

Le contexte d'appel est composé de trois instances reliées à deux domaines de protection différents :

- Le maçon lié au domaine étranger.
- Le frigo lié au domaine famille.
- Le contrôleur d'accès lié au domaine système.

Le domaine étranger n'a pas de permissions qui impliquent "donner bière", le contrôleur d'accès soulève une exception de sécurité.

Scénario 3 : le maçon demande à ma fille de prendre une bière dans le frigo

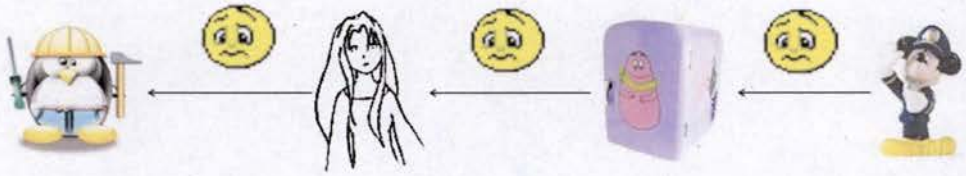

Figure : 10 Scénario 3 : le maçon demande à ma fille de prendre une bière dans le frigo

Le contexte d'appel est composé de quatre instances reliées à trois domaines de protection différents :

- Le maçon lié au domaine étranger.
- Ma fille liée au domaine famille.
- Le frigo lié au domaine famille
- Le contrôleur d'accès lié au domaine système.

Le domaine étranger n'a pas de permissions qui impliquent "donner bière", le contrôleur d'accès soulève une exception de sécurité.

Scénario 4 : ma fille demande au maçon de prendre un jus d'orange dans le frigo

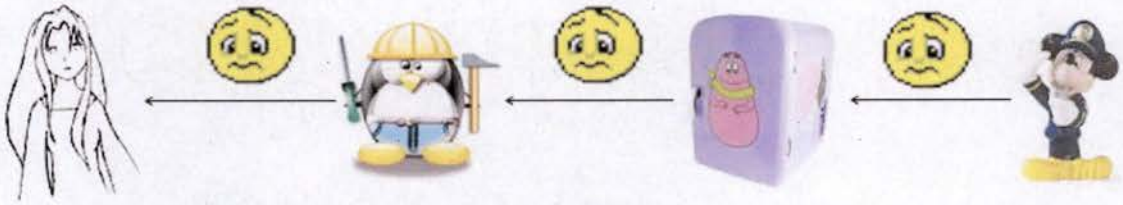

Figure 11 : Scénario 4 : ma fille demande au maçon de prendre un jus d'orange dans le frigo

Le contexte d'appel est composé de quatre instances reliées à trois domaines de protection différents :

- Ma fille liée au domaine famille.
- Le maçon lié au domaine étranger.
- Le frigo lié au domaine famille.
- Le contrôleur d'accès lié au domaine système.

Le domaine étranger n'a pas de permissions qui impliquent "donner jus d'orange", le contrôleur d'accès soulève une exception de sécurité.

3.2.5. Accès privilégiés

Ce type de contrôle est trop restrictif pour une série de règles d'accès.

Par exemple :

- Comment implémenter un comportement où ma fille peut donner une bière au maçon si le travail est terminé ?
- Comment donner la permission de connexion à une base de données à un composant sans donner cette permission au code utilisant ce composant ?
- Plus fondamentalement, comment permettre au *SecurityManager* d'accéder à la liste des permissions sans que le code appelant ne puisse lui-même y avoir accès ?

Action privilégié de l'AccessController

La classe `AccessController` possède les méthodes pour construire un contexte privilégié :

- `public static <T> T doPrivileged(PrivilegedAction<T> action)`
- `public static <T> T doPrivileged (PrivilegedExceptionAction <T> action)`

Les types `PrivilegedAction` ou `PrivilegedExceptionAction` sont des interfaces demandant l'implémentation d'un méthode `run()`.

Les appels de méthode à partir de la méthode `run()` de l'action se font dans un contexte privilégié. La vérification des permissions dans ce contexte ne tient compte que des domaines de protection des classes présentes "au-dessus" de `doPrivilged()`.

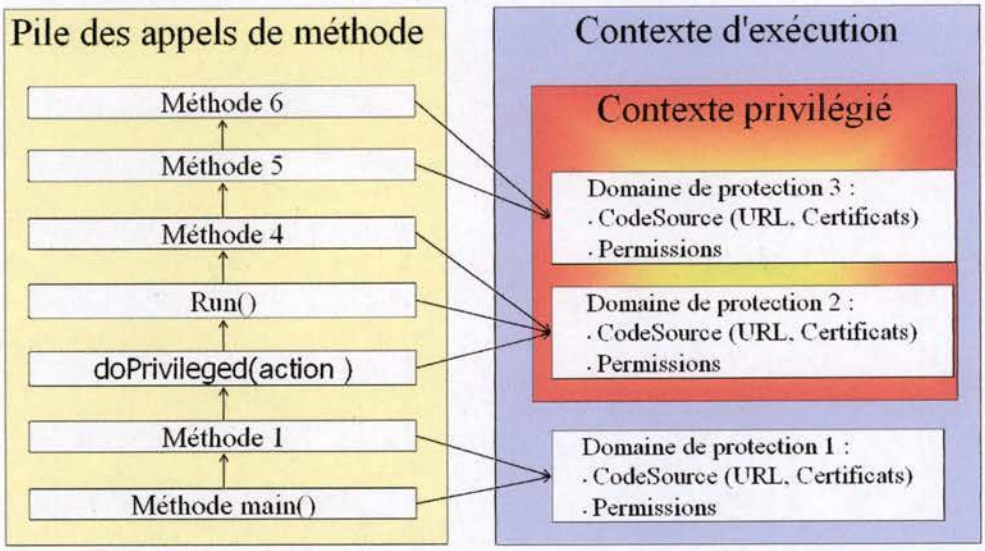

Figure 12 : Définition d'un contexte d'exécution privilégié en fonction de l'état de la pile d'exécution et de la position de l'appel à la méthode doPrivileged()

Illustration : accès privilégié au frigo sécurisé

Situation

Le maçon demande une bière à ma fille. Ma fille vérifie si le travail est terminé. Si c'est le cas, elle demande une bière au frigo et elle la donne au maçon.

Problème : la demande au frigo se passe dans un contexte où le maçon est présent, ce qui provoquera, suivant le modèle précédent, un accès refusé.

Solution : création par ma fille d'un contexte privilégié ne tenant pas compte du contexte d'appel de sa méthode.

Scénario 1 : le maçon demande à ma fille de prendre une bière dans le frigo

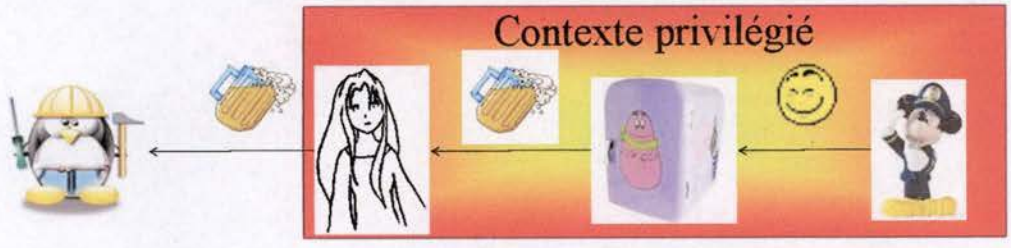

Figure 13 : Scénario 1 : le maçon demande à ma fille de prendre une bière dans le frigo

Le contexte d'appel privilégié est composé de trois instances reliées à trois domaines de protection différents :

- Ma fille liée au domaine famille.
- Le frigo lié au domaine famille.
- Le contrôleur d'accès lié au domaine système.

Le maçon lié au domaine étranger ne fait pas partie du contexte.

Tous les domaines du contexte privilégié ont des permissions qui impliquent "donner bière", le frigo donne la bière à ma fille.

scénario 2 : ma fille demande au maçon de prendre un jus d'orange dans le frigo

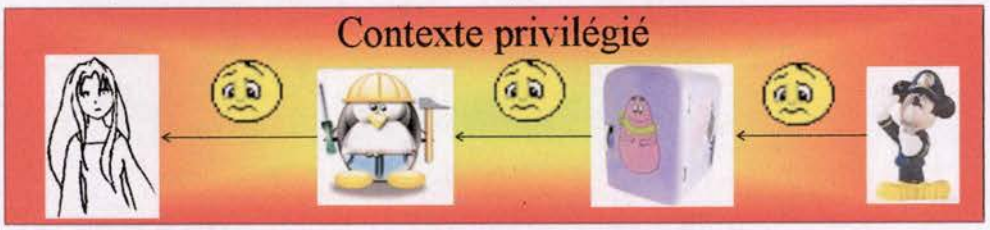

Figure 14 : scénario 2 : ma fille demande au maçon de prendre un jus d'orange dans le frigo

Le contexte d'appel privilégié est composé de quatre instances reliées à trois domaines de protection différents :

- Ma fille liée au domaine famille.
- Le maçon lié au domaine étranger.
- Le frigo lié au domaine famille.
- Le contrôleur d'accès lié au domaine système.

Le domaine étranger n'a pas de permission qui implique «"donner jus d'orange", le frigo soulève une exception de sécurité.

3.2.6. Exemple d'application de la politique de sécurité

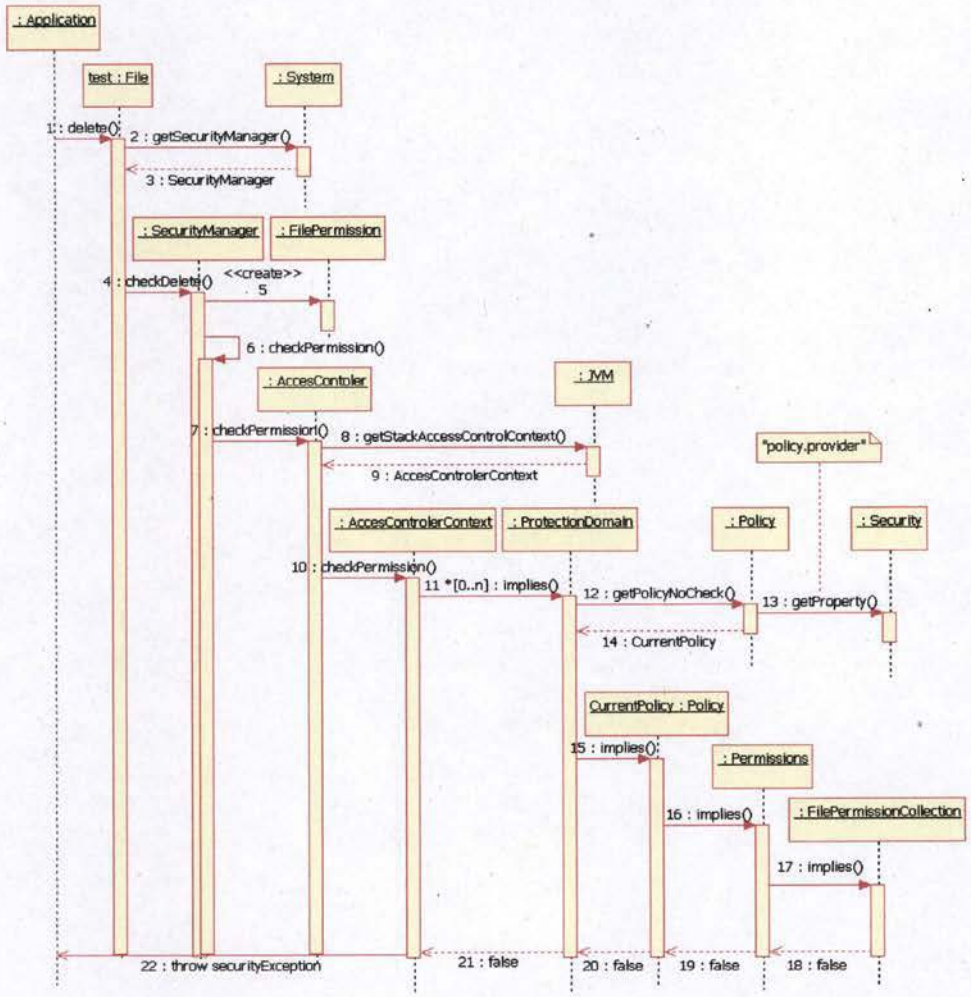

Figure 15 : Diagramme de séquence : refus de la suppression d'un fichier lors de l'appel de la méthode delete() d'un objet File

1 Une application appelle la méthode `delete()` d'un objet File précédemment initialisée avec le `pathName` : `"c:/tmp/test.txt"`.

2 La méthode `delete()` récupère le `SecurityManager` actif en faisant appel à la méthode `System.getsecurityManager()`.

3 la méthode `System.getsecurityManager()` retourne l'instance du `SecurityManager` actif.

4 Comme le `SecurityManager` n'est pas nul, la méthode `delete()` demande la permission d'exécuter l'action en appelant la méthode `checkDelete()` du `SecurityManager` avec comme paramètre `"c:/tmp/test.txt"`.

5 La méthode `checkDelete()`, pour tester la validité de l'action, crée une nouvelle `FilePermission` avec en paramètre le nom de fichier `"c:/tmp/test.txt"` et le nom de l'action `"delete"`.

6 La méthode `checkDelete()` délègue la vérification de la permission en appelant la méthode `chekPermission()` de son instance avec en paramètre la `FilePermission` créée à l'étape 5.

7 La méthode `checkPermission()` délègue la vérification de la permission en appelant la méthode statique `chekPermission()` de la classe `AccesControler` avec en paramètre la `FilePermission`.

8 La méthode `checkPermission()` de l' `AccesControler` récupère le contexte d'exécution en appelant la méthode native `getStackAccessControlContext()` .

9 la méthode native `getStackAccessControlContext()` retourne le contexte actif sous la forme d'un `AccesControlerContext`.

10 La méthode `chekPermission()` de l' `AccesControler` délègue la vérification de la permission en faisant appel à la méthode `checkPemission()` de l' `AccesControlerContext` en passant en paramètre la `FilePermission`.

11 La méthode `checkPemission()` de l' `AccesControlerContext` vérifie que la permission demandée est accordée par chacun de ses domaines de protection en appelant la méthode `implies()` de chacun d'entre eux avec la `FilePermission` en paramètre.

12 La méthode `implies()` d'un `ProtectionDomaine`, s'il a été construit en mode dynamique, récupère l'objet `Policy` actif grâce à la méthode statique `getPolicyNoCheck()`de la Classe `Policy`. Sinon, elle appelle directement la méthode `implies()` de son objet `Permissions` comme à l'étape 16.

13 La méthode `getPolicyNoCheck()`de la classe `Policy` crée l'instance de la `Policy` courante, si elle ne l'a pas encore été, en récupérant le nom de la classe à instancier en appelant la méthode statique de `Security`, `getProperty()`, avec `"policy.provider"` en paramètre.

14 La méthode `getPolicyNoCheck()` retourne l'instance courante de `Policy`.

15 La méthode `implies()` d'un `ProtectionDomaine` délègue la vérification de la méthode `implies()` à l'instance courante de `Policy` en appelant sa méthode `implies()` avec la `FilePermission` et sa propre référence en paramètre.

16 La méthode `implies()` de `Policy` initialise un objet `Permissions` (collection de `PermissionCollection`), en fonction des propriétés du domaine de protection et des permissions accordées pour ce domaine, puis délègue la vérification à cet objet en appelant la méthode `implies()` de celui-ci, avec en paramètre la `FilePermission`.

17 La méthode `implies()` de `Permissions` délègue la vérification à la `PermissionCollection` spécialisée pour les `FilePermission` en appelant la méthode `implies()` de celle-ci avec en paramètre la `FilePermission`.

18 La méthode `implies()` de la `PermissionCollection` retourne `false`.

19 La méthode `implies()` de `Permissions` retourne `false`.

20 La méthode `implies()` de `Policy` retourne `false`.

21 La méthode `implies()` de `ProtectionDomaine` retourne `false`.

22 La méthode `chekPermission()` de l' `AccesControler` soulève une `AccessControlException`.

3.3. Support pour la gestion de sockets sécurisés

Le support pour la gestion de sockets sécurisés (*Java Secure Socket Extension* JSSE [Sun Microsystem, Inc, 2006]) permet de sécuriser les communications réseaux utilisant TCP/IP. Il fournit une implémentation en Java des protocoles SSL et TLS en incluant :

- Le chiffrement des données.
- L'authentification du serveur.
- L'intégrité des messages.
- Et, optionnellement, l'authentification des clients.

En utilisant JSSE, les développeurs peuvent obtenir un transfert sécurisé de données entre un client et un serveur communiquant grâce un protocole de la couche application tel que HTTP, Telnet, ou FTP.

La gestion de la cryptographie utilisée par l'implémentation des protocoles sécurisés est fournie par le support JCA (*Java ™ Cryptography Architecture*). Les algorithmes sont fournis sous forme de *Provider* regroupant des classes implémentant les fonctionnalités concrètes de l'un ou l'autre algorithme.

La gestion des certificats et des clés privées, utiles au fonctionnement du protocole, sont confiées aux *KeyManagers* et aux *TrustManager*.

Les *KeyManager* offrent les fonctionnalités nécessaires pour accéder aux clés et certificats contenus dans une *Keystore* (base de données de clés).

Les *TrustManager* ont la responsabilité d'établir quels sont les certificats dignes de confiance ou non. Ils ont recourt pour cela à une *Truststore*.

Les *Keystore* sont des bases de données de clés concrètes. Elles sont utilisées notamment dans les processus d'authentification et d'intégrité des données. Elles possèdent deux types d'entrée :

- les entrées de type clé (*key entries*), constituées d'une entité d'identification (un certificat par exemple) et d'une clé privée.
- Les entrées de type certificat de confiance (*trust entries*) constituées uniquement d'une clé publique en plus de l'entité d'identification.

Les *Truststore* sont des *Keystore* utilisées par le *TrustManager* pour décider si une information est de confiance ou non. Grâce aux certificats des autorités de certification stockés dans la *Truststore*., il sera possible de déterminer si une information est signée par une clé garantie, par un certificat garanti et par une autorité digne de confiance.

3.3.1. Procédure de création d'une communication sécurisée.

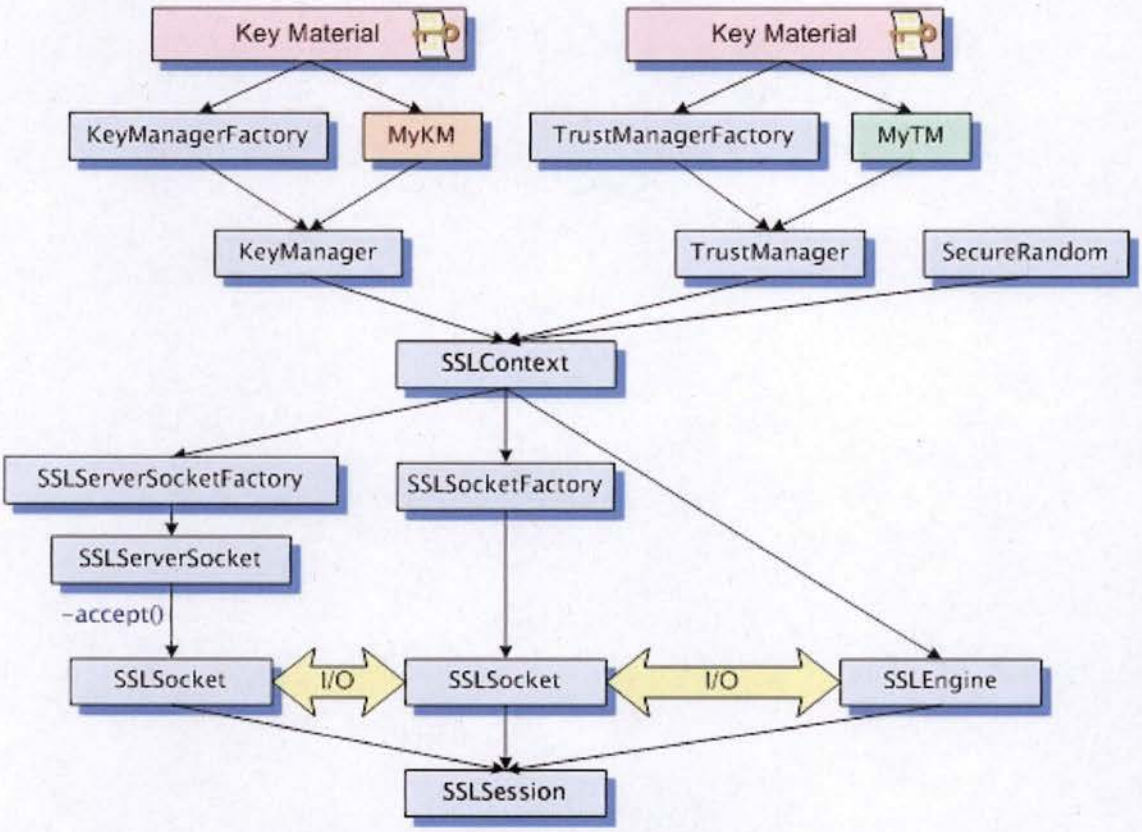

Figure 16 : Relations entre les classes de JSSE

Key Material

Les clés et les certificats sont stockés dans des *Keystore*. Dans la version de base fournie par Sun, les *Keystore* sont concrétisées par des fichiers au format « JKS » ou « PKS12 ». Chacun de ces formats est géré par des implémentations spécialisées de *KeyManager* ou de *TrustManager*.

La création de ces fichiers ainsi que leur administration peut se faire via un utilitaire en ligne de commande « keytool ». Cet utilitaire reprend les fonctionnalités suivantes :

- La génération de clés secrètes.
- La génération de paires de clé publique et privée.
- L'importation de certificats.

- La génération de requêtes pour la signature d'un certificat par une autorité de certification.
- L'exportation de certificats.
- Des fonctionnalités de gestion permettant de supprimer, protéger par un mot de passe, lister et visualiser les entrées de la *Keystore*.

Toujours avec l'implémentation de base fournie par Sun, le fichier utilisé par le *KeyManager* est celui dont la propriété système `javax.net.ssl.keyStore` précise la localisation sur le système de fichier. Le mot de passe utilisé pour accéder à la *Keystore* est précisé par la propriété système `javax.net.ssl.keyStorePassword`.

Par contre, le fichier utilisé par le *TrustManager* est celui dont la propriété système `javax.net.ssl.trustStore` précise la localisation sur le système de fichiers. Le mot de passe utilisé pour accéder à la *keystore* est précisé par la propriété système `javax.net.ssl.trustStorePassword`. Si ces propriétés ne sont pas définies au niveau du système, un de ces fichiers :

« `<java-home>/lib/security/jssecacerts` »

« `<java-home>/lib/security/cacerts` »

sera utilisé par défaut.

KeyManagerFactory et TrustManagerFactory.

Les deux fabriques *KeyManagerFactory* et *TrustManagerFactory* permettent de récupérer une instance de, respectivement, *KeyManager* et *TrustManager*, en fonction du nom d'un algorithme et d'un *Provider*. Ces fonctionnalités sont utilisées pour l'initialisation du contexte SSL (*SSLContext*).

Le nom des algorithmes à utiliser par défaut sont configurables dans le fichier « `<java-home>/lib/security/java.security` » grâce aux entrées :

« `ssl.KeyManagerFactory.algorithm` »

« `ssl.TrustManagerFactory.algorithm` »

SSLContext

Cette classe représente l'implémentation du protocole sécurisé. Elle agit comme une fabrique des *SSLSocketFactorys* et des *SSLEngines*. Elle regroupe tous les paramètres du protocole des instances créées à partir d'elles comme, par exemple, la liste des *CipherSuite* possibles ou si l'authentification du client est demandée.

SSLServerSocketFactory et SSLSocketFactory

Les deux fabriques *SSLServerSocketFactory* et *SSLSocketFactory* permettent de récupérer une instance de, respectivement, *SSLServerSocket* et *SSLSocket*.

Les sockets créés sont manipulables comme tout autre socket, mais toutes leurs entrées/sorties seront interceptées par le *SSLEngine* qui sera en charge de les sécuriser en fonction des paramètres du *SSLContext* dont il est issu. La différence entre un *SSLServerSocket* et un *SSLSocket* est que le serveur a la possibilité de se mettre en état d'attente d'une connexion d'un client via sa méthode `accept()`.

SSLEngine

Les instances de *SSLEngine* encapsulent les états et les opérations d'une connexion TLS/SSL sur les *buffers* d'entrées / sorties manipulés par ses utilisateurs. La figure ci-dessous illustre les flux de données entre un objet de la couche application et un objet de la couche transport passant par un *SSLEngine.*

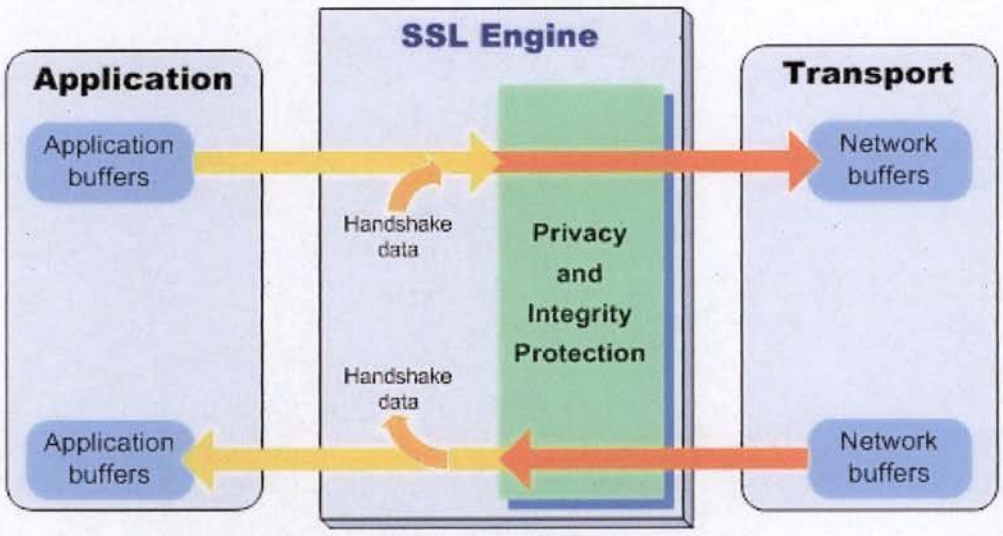

Figure 17 : Flux de données à travers un SSLEngine

Les messages concernant la phase de prise de contact (*handshake*) et la gestion du protocole sécurisé sont directement gérés par le *SSLEngine* de manière transparente pour l'application.

Les messages contenant des données en texte clair venant de l'application sont sécurisés par le *SSLEngine* avant d'être envoyés sous cette forme à l'objet transport.

Les messages de données venant de la couche transport sont validés et transmis à l'application sous forme de texte clair.

Tout *SSLEngine* peut être dans cinq état différents :

1 ***Creation*** : prêt à être configuré.

2 ***Initial handshaking*** : gestion de l'authentification et négociation des paramètres.

3 ***Application data*** : prêt pour l'échange de données avec l'application.

4 ***Rehandshaking*** : renégociation des paramètres.

5 ***Closure*** : prêt à couper la communication.

SSLSession

Une *SSLSession* représente le contexte négocié entre le client et le serveur d'une connexion SSL. La session contient la suite de chiffrement négociée dans cette connexion, les certificats d'authentification,... Elle contient aussi, mais de manière privée, la clé secrète négociée pour le chiffrement des données.

La *SSLSession* peut être récupérée auprès des *SSLSocket*.

3.4. Service d'authentification et d'autorisation JAAS

JAAS (*Java Authentication and Authorization Service*) [Sun Microsystems, Inc] est un framework de sécurité de Java. Depuis la version 1.4, il est intégré au JRE (environnement d'exécution Java). Il permet l'authentification des utilisateurs dans une application Java, et d'y associer des permissions (autorisations).

JAAS implémente une version Java de la norme *Pluggable Authentication Module* (PAM).

PAM est un mécanisme permettant d'intégrer différents schémas d'authentification de bas niveau dans une API de haut niveau. Ce mécanisme permet de rendre indépendante une application réclamant une authentification de la technique utilisée pour cette authentification.

L'administrateur système peut alors définir une stratégie d'authentification sans devoir recompiler des programmes d'authentification. PAM permet de contrôler la manière dont les modules d'authentification sont utilisés par les programmes par la simple modification d'un fichier de configuration.

Les programmes qui donnent aux utilisateurs un accès à des privilèges doivent être capables de les authentifier. Les techniques d'authentification sont diverses et dépendent généralement d'une politique et de techniques propres à l'entreprise. Il est donc important que les applications puissent facilement s'adapter.

3.4.1. Architecture

Les classes de l'architecture du service sont en fait un ensemble de classes et d'interfaces se répartissant en trois catégories :

- Les classes communes : `Subject`, `Principal`, `Credential`.

- Les classes et interfaces d'authentification : `LoginContext`, `LoginModule`, `CallbackHandler`, `Callback`.
- Les classes d'autorisation : `Policy`, `AuthPermission`, `PrivateCredentialPermission`.

Classes communes

La classe centrale de l'architecture de JAAS est la classe `javax.security.auth.Subject` qui regroupe les informations sur une personne sous la forme de `Principal` et de `Credential`, de portée publique ou privée.

Le `Subject` sera initialisé avec les informations le concernant durant la phase d'authentification.

Durant la phase de vérification, des autorisations seront accordées en fonction de ces informations.

Subject

Un *Subjet* possède un ensemble d'identités ou `Principal` qui le caractérise. Par exemple, un identifiant personnel et les identifiants des rôles qui lui sont associés. Il peut également posséder des propriétés de sécurité (`Credential`) publiques ou privées telles des clés de chiffrement.

Les méthodes permettant la manipulation de ces informations sont soumises à la vérification de permissions spécifiques. De plus, si une instance est positionnée en `readOnly`, il ne sera plus possible d'ajouter ou de retirer des informations de `Principal` ou de `Credential`.

Pour vérifier si un accès est autorisé en fonction d'un sujet connecté, la classe `Subject` possède 4 méthodes statiques :

- ```
  public static <T> T doAs( final Subject subject,
  final java.security.PrivilegedAction<T> action)
  ```
- ```
  public static <T> T doAs( final Subject subject,
  final java.security.PrivilegedExceptionAction<T> action)
  throws java.security.PrivilegedActionException
  ```
- ```
  public static <T> T doAsPrivileged( final Subject subject,
  final java.security.PrivilegedAction<T> action,
  final java.security.AccessControlContext acc)
  ```
- ```
  public static <T> T doAsPrivileged( final Subject subject,
  final java.security.PrivilegedExceptionAction<T> action,
  final java.security.AccessControlContext acc)
  throws java.security.PrivilegedActionException
  ```

Les méthodes `doAs()` et `doAsPrivileged()` permettent d'exécuter des actions privilégiées, dans le contexte d'un domaine de protection modifié avec les `Principal` du *Subjet* passé en paramètre. Tous les domaines intervenant à partir de l'appel de la méthode seront combinés avec ces `Principal`.

La différence entre `doAs`() et `doAsPrivileged`() se situe dans les domaines ajoutés aux domaines modifiés. `doAsPrivileged`() ajoute les domaines contenus dans l'`AccessControlContext` passé en paramètre. Avec `doAs`(), c'est toujours les domaines du contexte au moment de l'appel qui sont ajoutés (c'est-à-dire avant de tenir compte de l'utilisateur connecté).

Principal

Un `Principal` représente un identifiant lié à un sujet. Tout `Principal` doit implémenter l'interface `java.security.Principal`. Les `Principal` sont ajoutés aux domaines de protection pour déterminer les permissions de ce domaine.

Credential

Bien que tous les objets puissent être stockés comme `Credential`, il est conseillé qu'ils implémentent `javax.security.auth.Refreshable` et `javax.security.auth.Destroyable`.

Classes et interface d'authentification

Le processus d'authentification suit la chronologie suivante :

1 L'application instancie un `LoginContext`.

2 Le `LoginContext` consulte la configuration pour charger les `LoginModule` désignés par l'environnement d'exécution.

3 L'application appelle la méthode `login()` du `LoginContext`.

4 La méthode `login()` invoque les différents `LoginModule` pour qu'ils authentifient l'utilisateur. En cas de succès, chaque `LoginModule` ajoute des `Principal` et `Credential` au `Subject`.

5 Le `LoginContext` retourne l'état du processus d'authentification.

6 En cas de succès d'authentification, l'application peut récupérer le `Subject` auprès du `LoginContext`.

LoginContext

La classe `LoginContext` peut être instanciée avec plusieurs paramètres :

- Un nom obligatoire correspondant au nom qui sera recherché dans la configuration pour sélectionner les `LoginModule`.
- Une instance de `Subject`. Cette instance sera complétée par des `Principal` pendant la phase de connexion. Si ce paramètre n'est pas passé au constructeur, un nouveau `Subject` sera créé.
- un `CallbackHandler` permettant la communication entre les modules de d'authentification et l'application.

Elle implémente les méthodes permettant la gestion de l'authentification :

- `login()` pour provoquer le processus d'authentification des différents modules d'authentification
- `getSubject()` permettant de récupérer les informations sur l'utilisateur connecté.
- `logout()` pour déconnecter l'utilisateur. Les `Principal` sont retirés du `Subject`.

LoginModule

`LoginModule` est une interface permettant aux développeurs d'implémenter différentes techniques d'authentification pouvant être ajoutées à toute application utilisant un `LoginContext`.

CallbackHandler

Certains modules d'authentification ont besoin d'obtenir des informations des utilisateurs. Dans ce cas ils utilisent un `CallbackHandler`.

`CallbackHandler` est une interface avec une seule méthode :

```
void handle(Callback[] callbacks) throws java.io.IOException,
UnsupportedCallbackException
```

Le module prépare un tableau de `Callback` qui représente les demandes à l'application (par exemple, un `NameCallback` et un `PasswordCallback`) puis passe le tableau à la méthode `handle()` qui sera chargée de récupérer les valeurs auprès de l'utilisateur et de les stocker dans les `Callback` du tableau.

Callback

Le *package* `javax.security.auth.callback` contient les implémentations les plus courantes de `Callback` utilisées par les modules d'autentification.

Classes d'autorisation

Policy

Depuis la version 1.4 de Java, la classe concrète fournie par Sun, surchargeant la classe abstraite `java.security.Policy` , supporte des requêtes de vérification tenant compte des `Principal` référencés par un domaine de protection.

Le format de fichier de spécification de permissions intègre une syntaxe permettant d'accorder des permissions en fonction de `Principal`.

Partie II La conception du cours

4. Le contexte

4.1. Introduction

Le choix de cette thématique relève d'un constat établit à l'occasion du cours donné sur la sécurité dans le langage Java. Alors que dans les autres cours de programmation la motivation, l'intérêt et l'engouement des étudiants étaient quasi naturels. Face à ce cours, nous nous retrouvions devant des étudiants passifs voire bloqués. Il nous est donc apparu urgent et enthousiasmant de trouver un mode de transmission performant pour cette matière primordiale à leur futur professionnel.

Deux raisons majeures nous semblent expliquer le blocage des étudiants.

D'une part, le fait qu'un exposé uniquement théorique, reléguant l'apprenant à un rôle passif ne convenait pas à cette matière abstraite et ardue et, d'autre part, le fait que les apprenants ne semblaient pas percevoir l'utilité directe de cette matière pour leur vie professionnelle.

Rendre aux étudiants un rôle actif et directement productif en leur proposant des situations réalistes devenait pour nous une priorité.

4.2. La formation pour adultes

4.2.1. Apports théoriques

Au début, les recherches définissaient les méthodes d'apprentissage spécifique pour adulte par opposition avec celles pour enfant. Le terme d'andragogie s'opposait alors à celui de pédagogie, considérant que l'adulte qui apprenait était un être spécifique, différent de l'enfant.

Lindeman a établi dans « The meaning of adult education » [Lindeman 1926], les fondements d'une approche méthodologique de l'éducation des adultes. *« L'éducation des adultes sera envisagée sous l'angle des situations et non des programmes. Dans le système pédagogique traditionnel, c'est l'inverse, les principaux acteurs sont les programmes et les enseignants, les élèves n'étant que des éléments secondaires. [...] Le programme de formation pour adulte est conçu autour des besoins et des centres d'intérêts de ce dernier. »*

Héritier de ce mouvement, Malcolm Knowles s'inscrit dans la lignée de Lindeman et établit un modèle andragogique à l'antithèse du modèle pédagogique traditionnel de l'époque [Knowles 1973].

Selon l'auteur,

- L'apprenant adulte se distingue de l'enfant par son besoin de savoir pourquoi et comment il va entreprendre une démarche d'apprentissage.
- L'apprenant adulte est conscient de son autonomie et de sa capacité de libre choix, il veut donc être traité dans le respect de cette capacité d'autogestion. L'enfant, quant à lui, est un être dépendant de son professeur et de ses parents.
- L'apprenant adulte est déjà pourvu d'une identité propre et d'un bagage d'expériences qui constituent la plus riche ressource de l'apprentissage. Pour l'enfant la seule expérience utile est celle de son professeur.
- L'apprenant adulte aura plus de volonté pour apprendre si les compétences visées par la formation lui permettent de solutionner des situations réelles. Pour l'enfant, l'apprentissage de nouveaux acquis répond seulement à la logique de la matière.
- L'apprenant adulte trouve l'essentiel de sa motivation dans des pressions intérieures même si les motivations extérieures (meilleurs emploi,...) restent importantes, tandis que l'enfant est plus motivé par des stimulations externes comme les points, les récompenses,...

Cette vision négative de l'enfant, perçu comme un être dépendant, incapable de dresser des ponts entre un apprentissage académique et son expérience a été par la suite fortement critiquée et de nombreux travaux de Psychologie et de Sciences de l'Education ont considérablement réduit les différences entre les méthodes d'apprentissage des enfants et des adultes. Une différence majeure persiste cependant : l'expérience et, notamment, l'expérience socioprofessionnelle.

En France, Roger Mucchielli souligne l'importance de cette expérience chez l'adulte comme base d'une démarche d'apprentissage. Il démontre que les responsabilités familiales ou professionnelles, les rôles sociaux ou encore l'expérience directe de l'existence ont façonné un être plus pragmatique et plus conscient de ses limites et de ses potentialités. Ce réalisme va de pair avec une rigidification de leur personnalité, une plus grande résistance aux changements ainsi que des comportements défensifs répondant à des peurs telles que le jugement d'autrui, la sanction, la crainte d'une perte de capacité de mémorisation ou d'adaptation. L'auteur préconise alors, une formation qui valorise l'expérience, davantage axée sur la pratique concrète et réaliste. Selon lui, la formation doit s'effectuer dans un cadre sécurisant favorisant un apprentissage sans risque.

Enfin, les expériences menées au cours du siècle dernier par Kurt Lewin, psychologue américain, ont permis de développer des théories importantes pour les praticiens en dynamique de groupes.

Il montre, entre autre, que l'implication par la discussion est supérieure à l'écoute d'un exposé, qu'elle favorise la modification d'habitudes et réduit la résistance au changement. Selon Lewin, les interactions entre les personnes

dans un petit groupe atteint davantage l'individu qu'une communication de masse telle qu'une publicité ou une conférence.

4.2.2. Applications pratiques

La lecture de ces travaux nous ont permis d'épingler quelques axes forts et d'établir une méthode pédagogique adaptée à un public d'adultes. Voici les items que nous avons tenté de transposer dans nos formations.

- Cerner aux mieux les besoins des apprenants afin de proposer des actions de formation qui répondent à ces besoins.
- Placer les apprenants dans des situations réalistes et concrètes afin de stimuler leur intérêt pour l'apprentissage visé.
- Choisir des situations proches de la réalité des apprenants.
- Donner aux étudiants des responsabilités dans le processus d'apprentissage afin qu'ils soient acteurs et éviter ainsi qu'ils soient de simples récepteurs d'un savoir transmis par le formateur.
- Valoriser l'expérience de chacun et favoriser les échanges entre les apprenants en créant des mises en situation, des simulations ou des démarches de résolution collective de problèmes.

On le voit dans ces deux derniers points, le travail en équipe devient un moyen de susciter la motivation pour l'apprentissage ainsi qu'une valorisation de chacun devant le résultat obtenu.

Le cours présenté ci-dessous se base sur ces grandes lignes directrices pour établir un contexte aussi favorable que possible à l'apprentissage de cette matière. Il va sans dire que, dans la pratique, cela exige un travail d'évaluation et de régulation constante de la part du formateur qui veillera en permanence à la cohésion du groupe et s'assurera de la motivation de chacun.

4.3. La formation professionnelle

Ce cours de sécurité s'inscrit dans le contexte d'une formation professionnelle qualifiante de 6 mois, destinée à des demandeurs d'emploi. L'objectif est de former à la programmation Java.

STE-Formations Informatiques, l'organisme dans lequel j'exerce le métier de formateur depuis plus de 10 ans, organise des formations professionnelles dédiées principalement à la programmation.

Dans ce contexte, toute formation proposée se doit :

- D'être en adéquation avec les demandes techniques de l'entreprise.
- De rendre les apprenants directement opérationnels dans leur futur contexte professionnel.
- De développer leur capacité d'adaptation face aux changements, ce qui est essentiel dans un domaine tel que l'informatique.

- De développer leur productivité dans les tâches liées au métier.

La méthodologie appliquée, par un principe d'évaluation continue avec remédiations immédiates, vise clairement à dépasser les limites des apprenants tout en assurant une base solide visant à la pérennité de leur emploi.

4.3.1. Spécificité d'une formation professionnelle en programmation

Dans un domaine en constante mutation, proposant un nombre impressionnant de solutions différentes à un même problème, il est fondamental d'insister sur certaines compétences transversales qui permettront au futur employé de pouvoir s'adapter aux réalités du travail dans son entreprise. Ainsi, l'autoformation doit être mise en avant dans les différents cours dont celui de sécurité. Pour ce faire, il y a lieu de mettre en place des contextes d'apprentissage exigeant l'utilisation de différentes ressources comme la documentation officielle, les moteurs de recherches ou certains sites spécialisés. Pour pouvoir tirer parti de ces sources d'informations, l'apprenant devra nécessairement être capable d'analyser une demande et d'en extraire les points posant des problèmes de compréhension.

Un autre aspect récurrent dans le domaine de la programmation est le travail en équipe. Le programmeur doit être capable d'intégrer une équipe en s'y affirmant, en dégageant les tâches qu'il est à même d'accomplir et de communiquer de manière claire et concise le résultat de son travail. Le cours a également pour objectif de développer cette capacité.

Chaque société possède ses procédures, son organisation, sa solution de sécurité. Un nouvel employé se doit de pouvoir s'adapter rapidement à ces nouvelles données. Ce qui exige de sa part une bonne connaissance des concepts et principes inhérents à une problématique comme celle de la sécurité.

Enfin, une difficulté couramment rencontrée par un programmeur junior est de reprendre le développement initié par un autre programmeur. Chacun possède sa logique, sa façon d'appréhender un problème. Le programmeur junior manquant de pratique peut être déstabilisé par ces changements de méthode. Il est d'autant plus important de le confronter au code d'une tierce personne qu'il sera souvent cantonné à des travaux ponctuels de modifications au cours de son écolage en entreprise. Pour ce faire il devra faire preuve de ses qualités en logique de programmation pour reconstruire l'architecture du code à transformer. Le cours de sécurité envisagera également cette problématique.

4.4. Le public cible

En synthétisant, nous pouvons décrire le public cible comme suit :

- 12 adultes entre 20 et 40 ans.
- Niveaux d'études divers et variés : du secondaire supérieur, au master, dans des domaines pas forcément liés à l'informatique. On estime qu'une

moyenne de 30% des étudiants n'avaient jamais programmé avant d'entamer cette formation.

Certains d'entre eux ont connu l'échec scolaire et doivent se réconcilier avec le système éducatif. Pour la plupart, ils découvrent pour la première fois le principe de la formation d'adultes.

Ils sont, en général, animés d'une grande motivation à apprendre mais présentent également des centres d'intérêt techniques différenciés. Devant un panel aussi hétéroclite, il est important de veiller à la bonne progression de chacun en considérant leurs acquis respectifs, en replaçant leur progression individuelle au centre du processus et en assurant une valorisation de leurs réussites.

En ce qui concerne le cours de sécurité, nous sommes régulièrement confrontés à deux types d'à priori dans le chef nos étudiants :

- Les passionnés de techniques, qui ont déjà essayé d'aborder cette matière ardue par eux-mêmes sans succès, et qui considèrent que la sécurité est une grosse machinerie compliquée et ennuyeuse.
- Les rêveurs, qui se voient devenir hackers et qui sont fatalement déçus par notre cours qui n'a pas du tout l'ambition de les former à cette "discipline".

Le cours, en plus d'atteindre ses objectifs dans une matière difficile et de développer des compétences transversales essentielles se doit aussi d'être présenté de façon ludique, avec une pédagogie active afin de capturer l'attention de l'ensemble des apprenants.

4.5. Les pré-requis du cours

Ce cours intervient au deux-tiers de la formation. Les pré-requis techniques suivants sont souhaités :

- bonne logique de programmation,
- bonne connaissance du langage Java,
- bonne compréhension des concepts de la programmation objet,
- connaissance des protocoles réseaux,
- compréhension des diagrammes UML,
- connaissance des Design patterns les plus courants (dont MVC).

D'autres pré-requis sont également attendus :

- connaissance passive de l'anglais technique,
- capacité de réaliser un travail de synthèse,
- capacité de respecter des consignes, des délais.

Ces pré-requis ne sont pas toujours totalement rencontrés et le cours doit en tenir compte. Il sera pour eux l'occasion de les développer davantage.

Il existe régulièrement des différences de niveaux au sein de nos groupes de formation.

4.6. Temps imparti

Il est limité et c'est un autre facteur à intégrer lors de la préparation du cours. Nous disposons d'un total de 4 journées réparties en 8 demi-journées de plus ou moins 4 heures. Chacune de ces demi-journées représente une séance.

5. Conception du cours

5.1. Objectifs généraux

Vu l'étendue et la complexité de la matière à couvrir dans un laps de temps réduit, ce cours a été conçu de façon à donner aux apprenants les bases nécessaires pour pouvoir faire face à des demandes concrètes d'implémentation de spécifications de sécurité.

Il est important, pour atteindre cet objectif général, de développer chez les apprenants leur compréhension des principes de sécurité et des procédures actuellement utilisées pour sécuriser les systèmes informatiques. Il est également important, dans une formation professionnelle de programmeurs Java, qu'ils soient capables d'utiliser concrètement les différentes API proposées par le langage.

De plus, il est essentiel d'inscrire ce cours dans les objectifs généraux de la formation, à savoir :

- acquisition d'une bonne logique de programmation,
- apprentissage du langage et de l'API Java,
- maîtrise des principes de la programmation orientée objet,
- pratique d'UML et des Design Patterns,
- développement des capacités d'auto-apprentissage avec la documentation disponible,
- capacité à intégrer un code réalisé par un autre développeur,
- respect des consignes et des délais,
- capacité à travailler en groupe,
- rédaction de documents techniques et de synthèses.

5.2. Objectifs techniques spécifiques du cours

5.2.1. Comprendre les principes de la cryptographie

Objectif 1 Être capable de décrire les différences entre le chiffrement symétrique, chiffrement asymétrique et le hachage cryptographique.

Objectif 2 Pouvoir citer les principaux algorithmes de chiffrement utilisés pour chacune des techniques.

Objectif 3 Pouvoir expliquer les avantages et inconvénients du chiffrement symétrique et asymétrique.

Objectif 4 Être capable de définir : clé secrète, clé privée, clé publique, code d'authentification de messages, signature, certificat, autorité de certification.

5.2.2. Etablir une connexion sécurisée

Objectif 5 Pouvoir citer les différentes options de paramétrage d'une connexion sécurisée avec TLS/SSL.

Objectif 6 Pouvoir expliquer le rôle de chacune des techniques de chiffrement dans une communication sécurisée de type TLS/SSL.

Objectif 7 Être capable d'expliquer à quoi sont utilisés les éléments suivants dans une communication sécurisée TLS/SSL entre un client et un serveur :

- Le certificat de l'autorité de certification.
- La clé privée de l'autorité de certification.
- La clé publique de l'autorité de certification.
- Le certificat du serveur.
- La clé privée du serveur.
- La clé publique du serveur.
- Le certificat du client.
- La clé privée du client.
- La clé publique du client.

Objectif 8 Être capable de mettre en place une configuration permettant d'établir une communication SSL entre un client et un serveur Java, donc de réaliser :

- la mise en place d'une autorité de certification (*OpenSSL*),
- la génération d'une clé publique et privée avec les outils Java (« keytool »),
- la génération d'une requête certificat (« keytool »),
- la création du certificat garanti par l'autorité de certification (*OpenSSL*),
- l'ajout des certificats de l'autorité de certification(AC) et du serveur à la *keystore* serveur,
- l'ajout du certificat de l'AC à la *trusted keystore,*
- le démarrage des applications clientes et du serveur avec les arguments valides pour l'utilisation des *keystores*,

5.2.3. Gestion des permissions d'accès

Objectif 9 Etre capable de configurer la machine virtuelle pour l'utilisation du *SecurityManager* et des bases de données de permissions.

Objectif 10 Être capable d'accorder des permissions en fonction de l'origine du code.

Objectif 11 Être capable de créer un contexte privilégié pour n'accorder les permissions qu'aux codes qui en ont réellement besoin.

Objectif 12 Être capable de créer et d'utiliser des permissions surchargeant simplement `java.security.BasicPermission`.

Objectif 13 Être capable de créer et d'utiliser des permissions demandant la surcharge des méthodes des classes `java.security.Permission` et `java.security.CollectionPermission`.

5.2.4. Signature de code

Objectif 14 Être capable de créer un certificat permettant la signature de codes.

Objectif 15 Être capable de créer et de signer un fichier « jar » à l'aide des méthodes suivantes :

- en ligne de commande grâce à l'outil Java « jarsigner »,
- de façon automatique, dans une tâche « Ant ».

Objectif 16 Être capable d'accorder des permissions en fonction du fait que le code a été signé avec tel ou tel certificat.

5.2.5. Authentification

Objectif 17 Être capable de configurer la machine virtuelle pour utiliser JAAS, ce qui implique :

- la connaissance des clés dans le fichier « `java.security` » pour utiliser l'objet de configuration de base fourni par Sun,
- l'écriture d'un fichier de configuration de *login*,
- le paramétrage du lancement d'une application pour tenir compte du fichier de configuration en modifiant le fichier « `java.security` » ou en ajoutant un paramètre au lancement de l'application.

Objectif 18 Être capable d'implémenter les fonctionnalités nécessaires pour la communication entre un `LoginContext` et l'utilisateur final à l'aide de *Callback*.

5.2.6. Autorisation

Objectif 19 Être capable d'étendre `AccessControlContext` à partir du sujet connecté à l'application.

Objectif 20 Être capable d'accorder des permissions en fonction de l'utilisateur authentifié.

5.3. Méthodologie

L'organisation du cours doit amener les apprenants à avoir une démarche active dans la découverte et l'expérimentation de la matière. Il est conçu comme une mise en situation-problème proche de celles que l'on peut rencontrer en entreprise. Cette situation-problème a pour mission d'amener les étudiants à atteindre les différents objectifs du cours tout en leur laissant une marge de manœuvre dans le processus d'apprentissage. Le rôle du formateur ne sera plus de mener toute la démarche mais d'aider les étudiants à résoudre les défis posés par la situation et d'atteindre ainsi les objectifs du cours.

La mise en situation-problème est constituée :

- D'une application « prototype » opérationnelle que les étudiants devront manipuler pour y appliquer des consignes de sécurité.
- D'une liste de consignes.
- D'une série de questions ouvertes sur les principes généraux de la matière.

Le cours propose une alternance de périodes d'exposés théoriques, de recherches et d'exercices pratiques permettant l'implémentation de spécifications de sécurité.

Les apports théoriques ont comme objectif la compréhension des concepts et des principes généraux indispensables pour pouvoir comprendre la documentation technique de l'API Java.

Le travail de recherche se réalisera en groupes de 2-3 apprenants constitués par le formateur. Ce dernier aura pris soin de leur mettre à disposition une série de documentations techniques. Cette période sera suivie d'un travail de synthèse visant à fixer l'acquis sur un support accessible. Dans une optique socio-constructiviste, la synthèse sera réalisée avec un outil d'écriture collaborative (Wiki).

Lors de cette phase, le rôle du formateur sera de veiller à conserver une dynamique d'apprentissage positive, à veiller à la participation active de chacun ainsi qu'au respect des délais.

Une mise en commun, orchestrée par le formateur, finalisera ce travail. Le formateur pourra répondre aux questions éventuelles et, s'il le juge nécessaire, apportera un complément théorique.

Cette mise en commun sera également l'occasion pour le formateur d'évaluer la qualité du travail fourni par les différentes équipes.

La pratique intervient après chaque phase théorique et permet la consolidation de la compréhension des concepts. La pratique immédiate permet également une évaluation continue : dès qu'un problème survient, le formateur peut, soit expliquer à nouveau un point de matière à l'ensemble de la classe, soit faire une rémédiation individuelle.

La partie pratique est conçue pour :

- Vu le temps imparti, minimiser les tâches d'implémentation. L'application « prototype » étant opérationnelle, les apprenants peuvent se concentrer sur les implémentations directement liées à la sécurité.
- A l'aide de la liste de consignes et de questions ouvertes, favoriser une évolution autonome de chaque apprenant. Le formateur gardera du temps pour du travail de remédiation individualisée.
- Représenter les besoins récurrents du monde de l'entreprise, ce qui favorise leur degré de motivation.

Au terme de cette phase pratique, il sera demandé aux étudiants, via le wiki, de réaliser une synthèse des connaissances acquises durant la séance. Elle sera constituée du mode opératoire utilisé pour résoudre ce problème précis. Cette nouvelle ressource issue de l'expérience de tous les apprenants, constituera un outil extrêmement pratique dans leur futur contexte professionnel.

A noter que la matière envisagée dans ce cours fait l'objet d'une deuxième phase d'évaluation dans le cadre du travail pratique de la fin de la formation qui reprend la problématique de la sécurité.

5.3.1. L'ancrage

« La société Duke est une société de développement software. Elle a pour projet le développement d'une application réseau permettant à une personne suivant une formation d'avoir accès aux dossiers de cours sous forme de documents « html ». Les documents sont accessibles en fonction de la formation et de son curriculum. Le programme récupère les données relatives aux formations ainsi que leur curriculum auprès de centres spécialisés dans ce type de production. Pour chaque cours, elle lance une recherche sur un serveur de dossiers de cours pour récupérer le dossier approprié au cours.

La société de développement software Duke recherche une équipe de deux ou trois programmeurs auxquels elle confierait l'implémentation des aspects de gestion de la sécurité. Pour pouvoir faire un choix judicieux de ces collaborateurs, ils ont développé un petit prototype de leur application et demande aux équipes candidates de modifier ce prototype en y implémentant une liste de spécifications concernant la gestion de la sécurité. A cette liste de spécification est ajoutée une liste de questions permettant de guider les choix technologiques. Les équipes devront également remettre un rapport écrit reprenant les modifications effectuées, la justification de leur choix et la liste des documents de référence qu'ils estiment pertinents sur le sujet. »

5.3.2. Exposés théoriques

Les exposés théoriques seront proposés par le formateur en fonction des besoins exprimés dans le cadre de la situation-problème. Ils auront comme sujet :

- les objectifs de sécurité pour un système informatique,

- les types de cryptographie,
- les certificats numériques et le rôle des autorités de certifications,
- la signature électronique,
- le protocole de communication sécurisée TLS/SSL,
- les spécifications du langage Java contribuant à la sécurité,
- le concept de « bac à sable » en Java,
- l'architecture de l'API Java pour la gestion du contrôle d'accès aux ressources protégées,
- présentation du support Java pour la gestion des sockets sécurisés JSSE,
- présentation du service Java pour l'authentification et la gestion des autorisations JAAS.

5.3.3. Pratique

Une fois le choix posé de confronter les apprenants à une situation réaliste, il nous fallait choisir le contexte technique qui servirait à la fois de vecteur d'apprentissage théorique et d'environnement d'expérimentation technique. Comme mentionné plus haut, j'ai opté pour l'application « prototype » réalisé par mes soins. Pourtant, ce n'était qu'une des trois pistes permettant de remplir ces objectifs. Elles consistent en :

- créer un exercice simple pour chacun des objectifs,
- créer un exercice complet où les étudiants devront réaliser une application réaliste en y implémentant les règles de sécurité,
- fournir un code existant et opérationnel mais dégagé de tout souci de sécurité. Les étudiants y implémenteront les aspects de sécurité. Travail fait en groupe pour diminuer la charge individuelle.

Chaque méthode possède ses avantages et ses inconvénients.

L'exercice simple ciblé sur un point de matière permet à l'apprenant de se concentrer sur une thématique en se dégageant des autres. S'il vient à manquer une séance ou qu'il ne peut résoudre le problème posé, il peut raccrocher sans difficulté à la suivante puisqu'elles sont indépendantes. Cependant, ce morcellement de la matière implique des solutions simplistes éloignées de la réalité professionnelle. La motivation de l'apprenant risque de diminuer en ne percevant pas l'intérêt immédiat de l'exercice. De plus, la transposition dans un environnement réel risque de s'avérer complexe car des problématiques concrètes sont éludées.

Par exemple, ce code simple :

```
public class Test1{
  public static void main(String[] args){
    SecurityManager sm = System.getSecurityManager();
    if(sm != null){
      sm.checkPermission(new BasicPermission("message.test");
```

```
    }
    System.out.println("Mon message");
  }
}
```

Il permet de tester l'utilisation de l'API Java pour le contrôle d'accès à un cours. L'application affichera "Mon message" si la `BasicPermission("message.test")` est accordée par le domaine de protection de la classe `Test1`.

Par contre, il n'aborde pas du tout l'aspect de l'endroit où doit se trouver ce test pour s'assurer que l'action est bien protégée et ne peut être contournée.

Certains aspects de la sécurité en Java exigent des architectures complexes pour être pertinents.

L'exercice complet exige une programmation lourde, une phase de test et de validation conséquente avant de pouvoir aborder les aspects spécifiques de la sécurité. La difficulté se renforce dans les différentes phases de développement compliquant la localisation des erreurs, diminuant le temps que l'on pourra consacrer aux problèmes de sécurité à proprement parler. Le danger, ici, est de devoir augmenter le rythme pour maintenir ses objectifs au risque de brusquer certains, ou de donner des solutions toutes faites. Cela déposséderait l'apprenant de son travail, brisant sa motivation voire son estime de soi. Par contre, il apparaît évident que mené à bien, l'exercice complètement réalisé par l'apprenant reste la meilleure solution pour asseoir les acquis et renforcer le sentiment de dominer la matière. Le temps imparti rend la solution difficilement imaginable.

La dernière solution, celle que nous avons choisie, n'est pas pour autant dénuée de défauts.

Tout d'abord, un programmeur débutant éprouve quantité de difficultés à entrer dans un code qui n'est le sien. Peut-être plus encore que de l'écrire lui-même.

Ensuite, dans l'idéal, une application qui se veut sécurisée devrait toujours envisager la sécurité dès le début pour guider les choix d'architecture du code.

Par exemple, pour garantir qu'un accès à une ressource est toujours contrôlé, il est très important de prévoir une architecture forçant tous les codes y faisant appel à passer par la même méthode sans alternative possible. Sans une architecture de ce type, vérifier un accès protégé s'apparente à transporter de l'eau dans une passoire. Il y a toujours un trou que l'on a oublié de boucher.

Enfin, le travail de groupe demande d'être géré constamment par le formateur. On ne peut laisser une apprenant bloqué à une étape sans pouvoir passer à la suivante. Il risquerait de ne pas acquérir le reste de la matière et de bloquer le reste de son groupe.

Pourtant, ses mêmes défauts sont aussi source d'apprentissages :

- La capacité de rentrer dans le code de quelqu'un d'autre est une compétence transversale poursuivie dans l'ensemble de la formation et il n'existe pas d'autre moyen que la confrontation directe pour y parvenir. Cette réalité professionnelle est bien présente dans l'esprit des étudiants qui l'abordent souvent comme un défi ludique.
- Il est intéressant de constater les failles d'une architecture laxiste et les différents moyens de la contourner.
- Le temps important passé à résoudre un problème en se confrontant aux autres membres du groupe ou à toute la classe lors d'une mise en commun est compensé par le fait que ce savoir restera plus longtemps dans les esprits.
- Le travail en groupe provoque une émulation certaine ainsi qu'une valorisation individuelle.

La conception de l'application se devait d'être suffisamment complexe pour permettre d'aborder tous les objectifs mais de rester accessible dans un délai raisonnable. Elle devait aussi être conçue dans l'optique d'une sécurisation future. Tous ces aspects renforçant sa crédibilité d'un point de vue professionnel.

Par ailleurs, le choix d'une architecture basée sur des modèles permettait de fournir aux étudiants un exemple cohérent de l'utilisation d'un principe de modélisation avec lequel les étudiants éprouvent des difficultés lors de sa mise en pratique.

Le modèle « Fabrique Abstraite » permet de faire évoluer le projet en remplaçant l'instanciation d'une classe par celle d'une autre qui implémente la même interface.

Le Modèle « MVC » permet aux étudiants de s'épargner l'effort de compréhension de la partie visuelle de l'application. Ce modèle permet également de provoquer avec les étudiants une réflexion : dans quelle méthode est-il le plus pertinent de coder la vérification d'accès ?

L'implémentation du modèle « Presentation Model » permet que toutes les actions des utilisateurs aient un format commun, ce qui facilite la création de contexte personnalisé par des utilisateurs authentifiés pour l'exécution d'une action.

5.3.4. Consignes

Les listes de spécifications (consignes) et de questions sont conçues pour couvrir l'ensemble des objectifs du cours. Elles sont formulées de telle manière qu'une recherche sur les concepts sous-jacents aux termes utilisés soit nécessaire.

L'ordre des spécifications et des questions est établi en fonction d'une logique de progression dans la découverte de la matière. Les questions d'authentification et d'autorisation se situent après celles sur la gestion des

permissions. L'architecture Java est conçue de telle manière que la gestion des autorisations en fonction d'une authentification est une extension de l'architecture du « bac à sable » gérant le contrôle d'accès en fonction de l'origine du code.

Cet ordre n'est pas pour autant totalement rigide. Certaines parties sont indépendantes et peuvent être inversées. Cette indépendance permettra une plus grande autonomie d'organisation des étudiants. Cela donnera également la possibilité à un étudiant en difficulté par rapport à certains aspects ou absent lors d'une séance de pouvoir participer quand même à la suite du cours.

Les consignes à distribuer aux étudiants sont reprises dans le chapitre 8.

5.4. Déroulement du cours

Etape 1:

Choix du travail, de(s) sujet(s) de la recherche parmi l'offre proposée par la mise en situation.

Méthode: discussion et argumentation du groupe

Rôle du formateur: guide les apprenants en vue de respecter un ordre inhérent à certaines matières.

Durée: 10 min (à chacune des étapes, le formateur indique aux apprenants le temps imparti aux différentes tâches).

Etape 2:

Recherche de documents afin de permettre la réalisation de la tâche choisie parmi les outils de travail (Internet : documentation officielle java ou d'autres sites abordant la matière choisie)

Méthode: recherche par équipe de 3. Dans leur vie professionnelle future, les apprenants seront amenés à travailler en équipe.

Les apprenants inscrivent leurs résultats de recherche sur le Wiki.

Rôle du formateur: participe et évalue les moyens mis en place lors des recherches.

Durée: 20 min

Etape 3:

Mise en commun des résultats des recherches

Méthode: Chaque équipe fait part de ses premières réponses et de ses interrogations persistantes.

Les apprenants établissent, au cours des séquences, sur le Wiki un référentiel. Le référentiel comporte une liste de sites de références qui servira de clé pour la réalisation de futures tâches similaires.

Rôle du formateur: lors la confrontation des résultats, il fait ressortir la pertinence des informations.

Durée: 20 min

Etape 4:

Recours à une personne ressource (formateur) afin de répondre aux interrogations persistantes.

Méthode: Sur base des interrogations apparues durant l'étape 3, le formateur fait part au groupe des notions nécessaires et explicite celles-ci afin de permettre de répondre aux interrogations persistantes.

Durée: 40 min

Etape 5:

Rédaction des réponses ou Codage

Rédaction des réponses

Méthode: répartition des tâches de travail par équipe de 2 ou en solo.

Il est important pour les apprenants de parvenir au mieux au partage des tâches car ils y seront confrontés dans leur vie professionnelle.

Durée: en fonction de la tâche à réaliser.

Rôle du formateur: évalue le niveau de compréhension de la matière de chacun et donne des apports individuels supplémentaires si le besoin se présente.

OU/ET

Codage

Méthode: travail en équipes de 2

Rôle du formateur: Evalue le niveau des compétences de chacun et est présent auprès des équipes ou du groupe en tant que personne ressource.

Durée: en fonction de la tâche à réaliser.

Etape 6:

Evaluation du travail

Méthode:

Evaluation en commun des réponses apportées et insertion de ces dernières dans le référentiel.

OU/ET

Comparaison des codages réalisés par les différentes équipes. Un programmeur est amené à rentrer dans la logique des codages d'autrui.

Si des réponses sont toujours en suspens, elles seront réintégrées au choix des séquences ultérieures lors de l'étape 1. Sauf, si le groupe estime qu'il a en mains les pistes nécessaires pour y répondre grâce au référentiel mis en place.

Durée: 40 min

5.4.1. Exemple de déroulement d'une séance

Rappelons tout d'abord que les étudiants ont reçu le premier jour un ensemble de documents. Une liste des matières à envisager dans le cadre du cours. Pour chaque matière, un ensemble de questions portant sur des points théoriques ainsi qu'une liste d'implémentation à réaliser.

En début de séance, les étudiants avec le formateur décident d'aborder la matière concernant la gestion du contrôle d'accès. Vu le nombre de tâches à réaliser pour ce point, *ils* décident de n'en aborder qu'une partie. Le formateur leur propose de choisir entre une approche où l'on aborde toute la théorie lors de cette séance pour laisser la pratique pour la séance prochaine, ou bien une démarche où on allie pratique et théorie.

Les étudiants choisissent d'allier pratique et théorie et d'aborder les trois premières questions théoriques et les deux premières implémentations de ce thème.

La proposition est validée par le formateur car les réponses aux questions théoriques choisies sont suffisantes pour aborder les implémentations retenues.

Une fois les équipes formées, chacune d'entre elles dresse la liste des informations indispensables en vue de la réalisation de la tâche. Ce travail est réalisé à l'aide des mots-clés utilisés dans les questions et l'énoncé des implémentations fournies sur ce point. Les références aux pages Internet trouvées seront ajoutées à la liste.

Dans le cas de la gestion du contrôle d'accès, les étudiants devront trouver de la documentation sur :

- les classes de l'API citées dans les questions (javadoc, *Security Architecture* dans la documentation Java, « bac à sable » et la matière relative au contrôle d'accès sur « Wikilabus[1] », ...).
- La définition des paramètres de lancement de la machine virtuelle pour utiliser le *SecurityManager*.
- La définition des tâches « Ant » pour le paramétrage du lancement d'une application Java.

On peut facilement imaginer que lors de la mise en commun, des demandes d'explications techniques sur le fonctionnement du *SecurityManager* Java soient exprimées.

Si tel est le cas, le formateur décidera de faire un exposé sur les principes du « bac à sable » et du domaine de protection.

De toute façon, il terminera son exposé en donnant des pistes pour compléter les informations techniques manquantes. Elles leur seront utiles dans l'implémentation future.

Après cette phase d'exposé, le travail de rédaction des réponses aux questions théoriques est réparti entre les groupes. Il est réalisé à l'aide du Wiki.

Ensuite, chaque groupe travaille à l'implémentation de toutes les consignes liées au point de matière choisi.

Le formateur se déplace de groupe en groupe pour aider en fonction des problèmes rencontrés.

En fin de séance, une mise en commun portera sur la validation à la fois des réponses aux questions théoriques et des implémentations pratiques. Elle sera faite par les groupes eux-mêmes.

Le formateur demandera à un des groupes de présenter sa solution de modification du fichier « buid.xml » pour le démarrage de l'application sous la responsabilité du *SecurityManager*. Il demandera à un autre sa version du fichier « ecole.policy ». Les deux solutions seront discutées par les autres groupes dans la mesure où ils ont une solution différente.

---

[1] Wikilabus est un site ou les formateurs et les stagiaires de STE-Formations partage des notes de cours de manière collaborative.

6. Description technique de l'exercice intégré

L'application à réaliser servira de base à la mise en pratique des techniques de sécurisation d'une application en Java. Elle devra permettre l'expérimentation concrète des différents concepts et techniques repris dans les objectifs du cours.

Vu le temps imparti pour le cours et de façon à concentrer l'apprentissage sur les mécanismes de sécurité, une application sans spécification de sécurité est donnée aux étudiants. Seules les spécifications de sécurité seront à implémenter par les étudiants.

6.1. Cahier des charges de l'application de base

Pour pouvoir expérimenter la mise en place d'une connexion sécurisée :

1 L'application sera construite sur le mode client/serveur.

2 Une connexion ce fera au niveau socket (pour expérimenter les techniques de base).

3 Une connexion http devra aussi être possible pour expérimenter une implémentation du protocole https.

Pour pouvoir expérimenter la gestion des permissions d'accès, l'application devra :

1 Permettre la vérification de permissions simples où seule l'utilisation existante dans l'API Java est nécessaire.

2 Permettre la vérification de permissions demandant l'implémentation des méthodes de `java.security.permission` et de `java.security.CollectionPermission`.

3 Charger ses classes à partir de différentes origines.

4 Offrir la nécessité de devoir créer des contextes privilégiés pour limiter les permissions d'accès aux codes sources qui en ont la responsabilité.

La signature de code et le déploiement en plusieurs fichiers « jar » sera automatisée.

Pour faciliter l'implémentation de l'authentification des utilisateurs, les classes gérant l'interface homme / machine seront implémentées mais non utilisées dans la version de base.

Pour faciliter l'implémentation de la gestion des permissions en fonction de l'utilisateur, les événements de l'interface graphique seront écoutés par une classe unique représentant l'utilisateur connecté.

6.2. Présentation de l'application à sécuriser

Le thème de l'application a été choisi pour correspondre à un contexte métier familier à tous les étudiants en formation : la mise en ligne de dossiers de cours en fonction des formations offertes par un centre de formation.

L'application est divisée en deux composants principaux :

- Une application serveur distribuant à la demande les dossiers de cours sous format « html ».
- Une application cliente permettant de visualiser le contenu d'un dossier de cours en fonction d'un choix d'une formation et d'un de ses cours.

6.2.1. L'application serveur

Le seul objectif pédagogique de l'application serveur est de pouvoir expérimenter la mise en place d'une connexion sécurisée avec le protocole SSL [objectif 8].

Son implémentation et son interface ont été réduites au maximum. La connexion au serveur se fait directement au niveau socket avec un protocole propriétaire fort simple. L'interface graphique permet uniquement d'arrêter le serveur.

L'instanciation du socket serveur a été isolée dans une méthode privée qui sera la seule à devoir être modifiée pour transformer la connexion non sécurisée en connexion sécurisée.

Le serveur devra utiliser une connexion sécurisée avec SSL lorsqu'il est démarré avec, comme paramètre, « -ssl ». Cette fonctionnalité n'est pas présente dans l'implémentation de base. Elle le sera lors de l'expérimentation des connexions sécurisées.

Figure 18 Interface du serveur de dossiers de cours

6.2.2. L'application cliente

L'application cliente a été conçue en fonction des objectifs d'expérimentation pratique de l'implémentation de sécurité.

L'interface graphique et les messages en console ont été conçus pour donner un feedback immédiat concernant le processus d'application de la politique de sécurité.

- L'application est exécutable avec ou sans l'application de la politique de sécurité [objectif 9]. Ceci permet également aux étudiants de déterminer si

les comportements d'erreur sont liés à l'application de la politique de sécurité ou à une erreur d'implémentation.

- Le code exécutable de l'application est réparti dans différents fichiers « jar » ce qui permet d'expérimenter l'attribution des permissions en fonction de l'origine du code [objectif 10] et la création de contextes privilégiés [objectif 11].

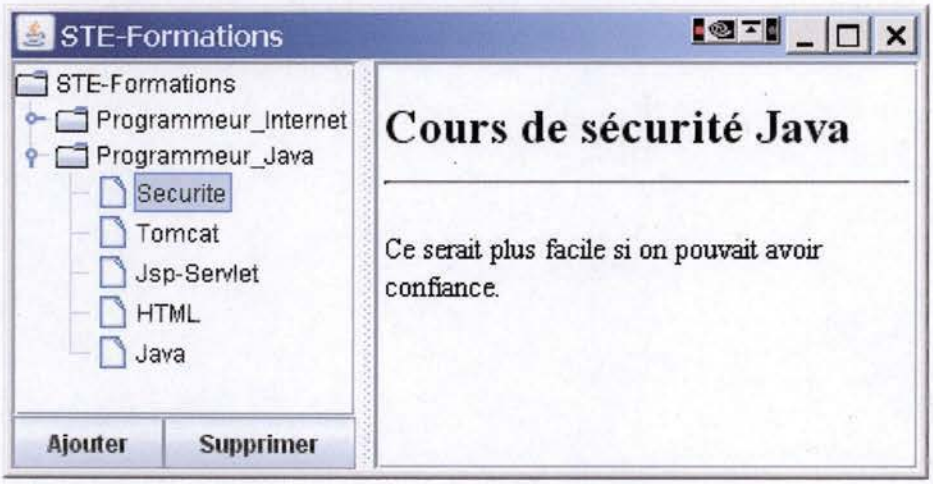

Figure 19 Interface de l'application cliente sans gestion de la sécurité

- Le fait que les messages d'erreur des exceptions soient systématiquement redirigés vers la console permet aux étudiants de visualiser le contexte d'exécution au moment de l'erreur [objectif 10].
- La politique de sécurité devra permettre de préciser les formations pour lesquelles l'accès à la liste des cours est autorisé. Sur le même modèle, l'accès au dossier d'un cours devra aussi pouvoir être contrôlé [objectif 12].
- La politique de sécurité devra permettre de préciser si l'ajout ou la suppression de cours est permise pour une formation donnée [objectif 13].

6.3. Organisation du code en sous-systèmes

Pour pouvoir expérimenter l'attribution des permissions dans un contexte de code distribué, les responsabilités de l'application cliente ont été réparties dans quatre sous-systèmes distincts. Chaque sous-système est déployé dans un fichier « jar » différent.

- Le sous-système « net » est chargé de la connexion au serveur.

  Il offre ses services sous la forme d'instances de *java.net.URL* construites à partir d'un nom de cours.

  Le code de l'API Java utilisé par ce sous-système a besoin de 2 permissions :

  - `java.net.SocketPermission`

- `java.net.NetPermission`

Cette situation permet d'expérimenter :

- Les différences d'exécution du code quand la machine virtuelle est démarrée sous le contrôle d'un *SecurityManager*.
- La technique d'attribution des permissions via le fichier des permissions.
- La création d'un contexte sécurisé permettant de n'accorder les permissions qu'au `CodeSource` du fichier « jar » dans lequel est déployé le sous-système.

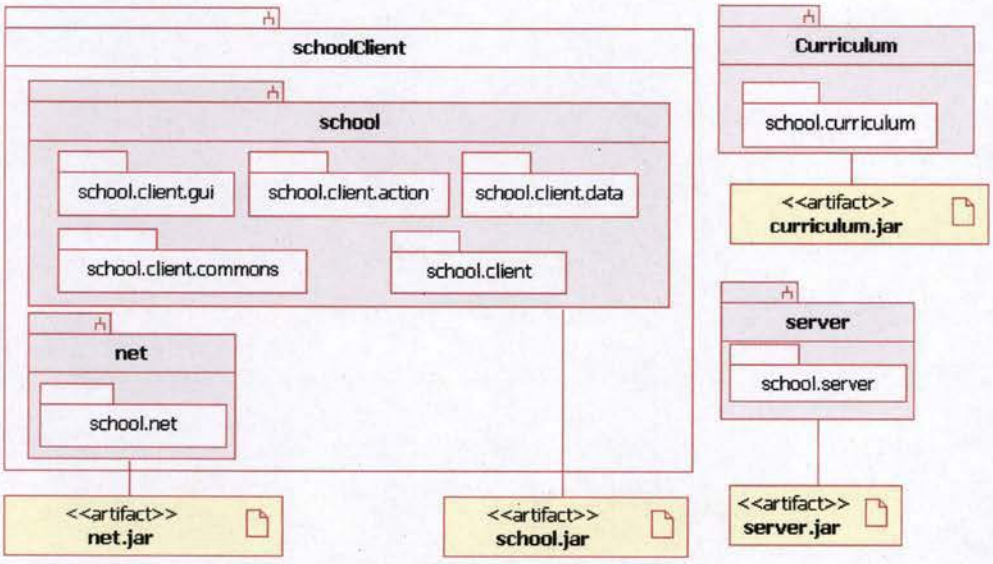

Figure 20 Organisation du code dans les sous-systèmes

- Le sous-système « curriculum » est chargé de distribuer la définition des curriculums de formation disponibles pour une école. A partir du nom de l'école, il crée et initialise des instances de `school.client.commons.Curriculum`.

Pour assumer cette responsabilité le sous-système a besoin d'un accès à un fichier texte situé dans le même fichier « jar » que le code source. La politique de sécurité permet toujours à un code d'accéder aux ressources situées dans le même fichier que celui dont il provient, mais comme la lecture du fichier est exécutée dans le contexte d'un appel de méthode venant du sous-système « school », il faut :

- soit accorder une `java.io.FilePermission` au `CodeSource` correspondant au fichier « school.jar »,
- soit exécuter la lecture du fichier dans un contexte privilégié.

Cette situation permet d'expérimenter ces deux scénarios et d'illustrer le cas particulier d'accès à de ressources se situant au même endroit que le code.

- Le sous-système « school » est le cœur de l'application cliente. Il contient la méthode main principale.

  Le code est organisé selon le design pattern « Modèle Vue Contrôleur » (MVC). La maîtrise de ce type d'organisation de code est un objectif transversal de la formation. Il est donc important de le mettre en place dans cet exercice.

  De plus, cette organisation du code permet de situer de manière logique la vérification des permissions au niveau du modèle qui est en charge de donner un accès et de modifier les données. Un code organisé rigoureusement suivant ce modèle ne permet un accès aux données qu'à travers les classes qui les représentent. La vérification des permissions à ce niveau empêche tout accès non vérifié aux ressources protégées.

  Par contre, dans le cadre d'une utilisation de l'application par un utilisateur reconnu par le système, la création du contexte privilégié d'exécution pourra se faire avant l'appel au contrôleur. De cette manière, tous les appels de méthodes nécessaires à l'exécution d'une action initiée par l'utilisateur, se fera dans le cadre de la politique de sécurité liée à cet utilisateur.

Pour faciliter la compréhension de l'application et de sa structure, les classes ont été réparties en package en fonction de leur rôle :

- Les classes jouant le rôle de vues sont regroupées dans le package `school.client.gui`.
- Les classes jouant le rôle de contrôleurs sont regroupées dans le *package* `school.client.action`.
- Les classes jouant le rôle de modèles sont regroupées dans le *package* `school.client.data`.

Les deux autres *package* se définissent comme suit :

- Le *package* `school.client` ne contient que la classe contenant la méthode main.
- Le *package* `school.client.commons` regroupe une fabrique abstraite avec les interfaces que les méthodes de la fabrique retournent. La fabrique abstraite a pour responsabilité d'encapsuler l'instanciation des classes concrètes. Toutes les instanciations d'une classe passent par la fabrique

6.4. Automatisation du déploiement et de l'exécution du code

Le code est accompagné d'une procédure de déploiement gérée avec l'outil de déploiement automatisé « Ant ». Cette procédure est présentée sous la forme d'un fichier texte que les étudiants devront améliorer progressivement durant le cours.

Les techniques de sécurité abordées par les objectifs sont en grande partie dépendantes de la manière dont est déployée l'application et des paramètres de lancement de l'application :

- Le fait que les accès soient vérifiés ou non par le *SecurityManager* dépend des paramètres de lancement.
- L'emplacement des fichiers « jar », des fichiers de permissions, des « keytool » et du fichier de configuration pour l'authentification sont à préciser comme paramètres au lancement de l'application.
- La répartition des différentes classes dans les fichiers de déploiement et la signature de ces fichiers sont des tâches de déploiement dont dépend la validité des tests faits sur l'application.

Dans ce cadre, l'automatisation de ces tâches apporte d'une part une diminution importante des risques d'erreurs de déploiement qui compromettraient la validation des tests effectués par les étudiants et d'autre part, minimise le côté fastidieux de la tâche qui risque de freiner l'envie de tester les modifications de codes produits par les étudiants. Ce dernier point est capital en période d'apprentissage.

Par contre, l'utilisation de l'outil « Ant » demandera aux étudiants un certain temps d'adaptation dont il faudra tenir compte dans le déroulement des cours.

Partie III Documents à distribuer aux étudiants

7. Description de l'application prototype

7.1. Cas d'utilisation

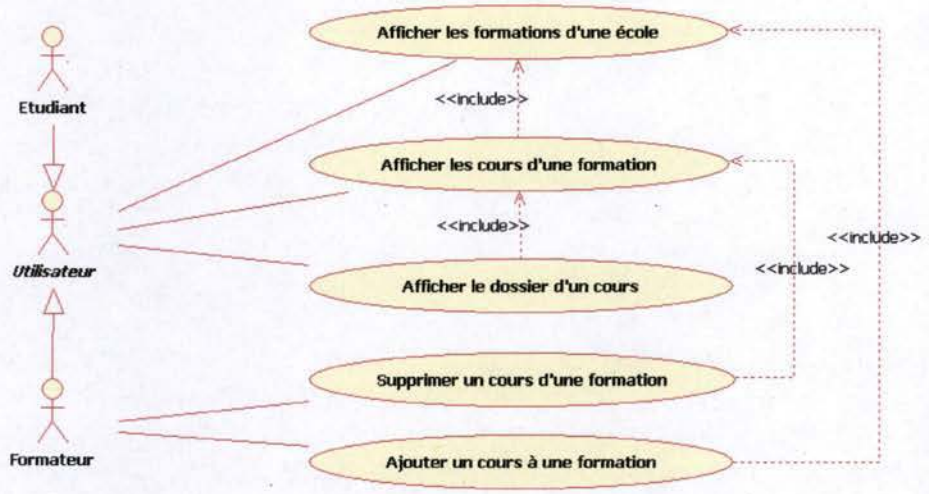

Figure 21 : Diagramme des cas d'utilisations

7.1.1. Cas 1 : afficher les formations d'une école :

Pré-condition : le système affiche la liste des écoles disponibles.

Scénario :

- L'utilisateur choisit une école et demande l'affichage de ses formations.
- Le système affiche les formations de l'école choisie.

Post-condition : la liste des formations de l'école choisie par l'utilisateur est affichée.

7.1.2. Cas 2 : afficher les cours d'une formation :

Pré-condition : la liste des formations d'au moins une école est affichée (cas 1).

Scénario :

- L'utilisateur choisit une formation et demande l'affichage de ses cours.
- Le système affiche les cours de la formation choisie.

Post-condition : la liste des cours de la formation choisie par l'utilisateur est affichée.

7.1.3. Cas 3 : afficher le dossier d'un cours

Pré-condition : la liste des cours d'au moins une formation est affichée (cas 2).

Scénario :

- L'utilisateur choisit un cours et demande l'affichage de son dossier.
- Le système affiche le dossier du cours choisi.

Post-condition : le dossier du cours choisi par l'utilisateur est affiché.

7.1.4. Cas 4 : supprimer un cours d'une formation

Pré-condition : la liste des cours d'au moins une formation est affichée (cas 2).

Scénario :

- L'utilisateur choisit un cours et demande sa suppression.
- Le système supprime le cours de la liste des cours et affiche la nouvelle liste.

Post-condition : le cours supprimé n'appartient plus à la liste des cours de la formation à laquelle il appartenait.

7.1.5. Cas 5 : ajouter un cours à une formation

Pré-condition : la liste des formations d'au moins une école est affichée (cas 1).

Scénario :

- L'utilisateur choisit une formation et demande l'ajout d'un cours.
- Le système demande le nom du cours à ajouter.
- L'utilisateur encode le nom du cours et valide son choix.
- Le système ajoute le cours à la liste des cours et affiche la nouvelle liste.

Post-condition : le cours encodé par l'utilisateur est ajouté à la liste des cours de la formation choisie par l'utilisateur.

7.2. Architecture de l'application

L'application est divisée en trois sous-systèmes représentant chacun un type de partenaire du projet.

1 Le serveur de dossier de cours : ce système est destiné à être déployé chez les partenaires proposant l'accès en ligne à des dossiers de cours.

2 Curriculums de formation : ce système est fournit par des partenaires spécialisés dans la conception et l'évaluation de curriculums de formation. La distribution des curriculums est faite sous la forme de codes distribués et maintenus par ces partenaires.

3 L'école : liant les cours proposés dans les curriculums aux dossiers de cours mis en ligne sur les serveurs de dossiers.

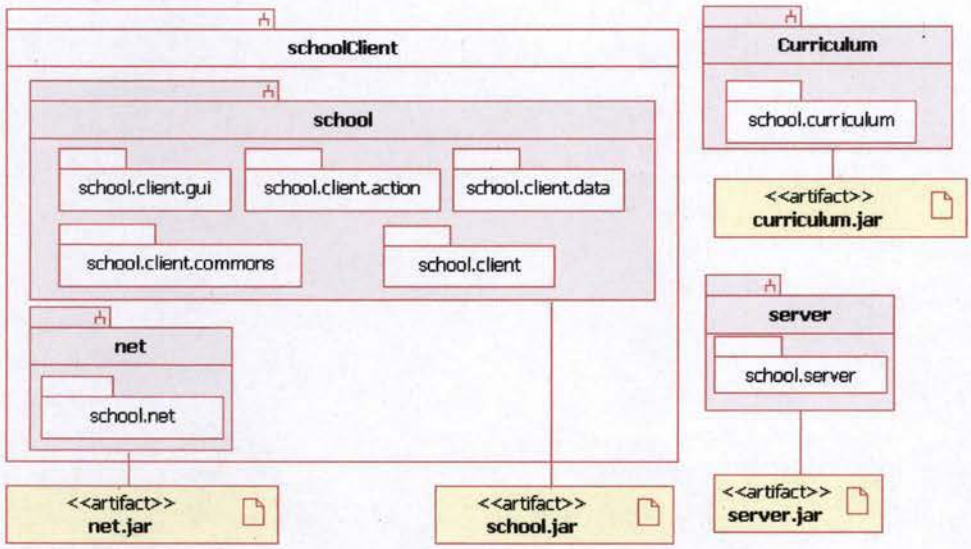

Figure 22 : Organisation du code de l'application prototype par sous-systèmes

7.2.1. Serveur de dossiers

Le serveur de dossiers communique avec l'école au travers d'un protocole réseau propriétaire élémentaire (protocole « book ») travaillant directement avec le socket proposé par Java. Dans la version actuelle, les dossiers de cours sont de simples pages « html ».

7.2.2. Curriculum

Le sous-système « curriculum » est conçu comme un code extérieur au système « school ». Les partenaires produisant des curriculums de formation fournissent des classes implémentant les interfaces :

- `school.client.commons.CurriculumFactory`
- `school.client.commons.Curriculum`

7.2.3. School

Le sous-système « school » est le cœur de l'application. Il est divisé en deux sous-systèmes déployés séparément afin d'isoler les classe gérant le protocole « book » du reste de l'application.

Le sous-système « net » encapsule les particularités du protocole en fournissant à travers une `school.client.commons.NetFactory` une `java.net.URL` personnalisée à partir d'un nom de cours.

Le sous-système « school » gère l'interface graphique grâce au pattern MVC (Modèle, Vue, Contrôleur) et la liaison entre le nom du cours et le dossier à récupérer sur le serveur. Pour pouvoir faire évoluer le projet, toutes les instanciations de classes sont de la responsabilité de fabrique (*Factory*).

Chaque *package* regroupe des classes de même rôle et une fabrique concrète permettant de les instancier :

- `school.client.action` : regroupe les classes « Contrôleurs » des actions utilisateurs.
- `school.client.data` : regroupe les classes « Modèles ».
- `school.client.gui` : regroupe les classes de gestion des « Vues » de l'application.
- `school.client.commons` : reprend les interfaces communes implémentées par les autres *packages* ainsi que la fabrique générale encapsulant les autres.

7.3. Modèle, vue, contrôleur avec modèle de présentation

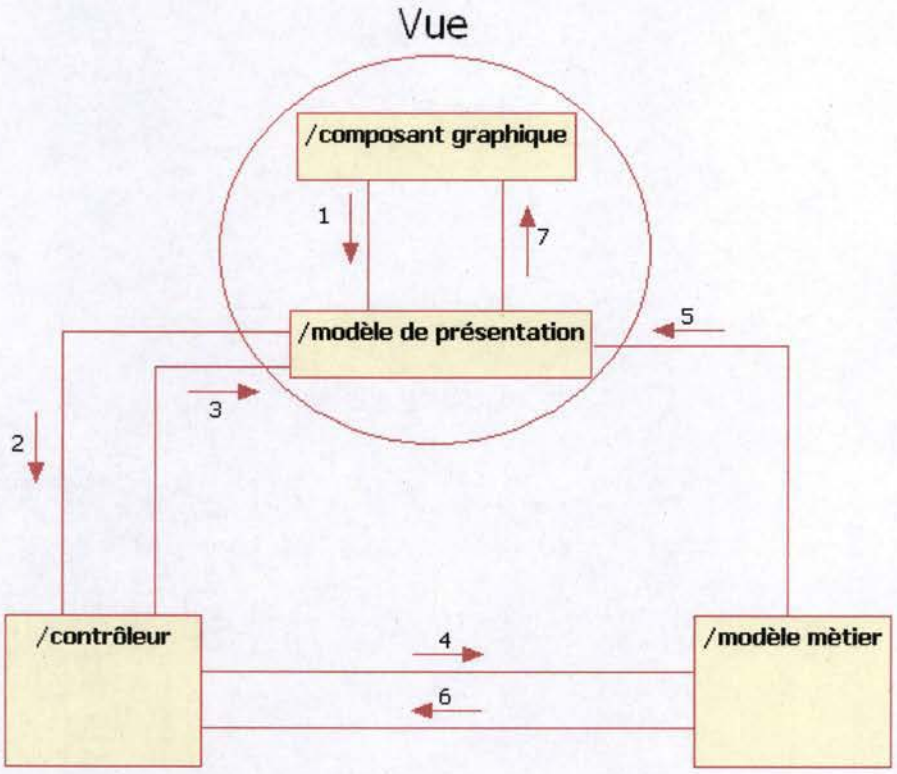

Figure 23 : Diagramme de collaboration entre les différents rôles des classes étudiant à l'architecture MVC

La vue est dans cette architecture divisée en deux types de classe ayant des rôles différents :

- Composant graphique : son rôle est de visualiser le modèle de présentation :
  - Il transmet les événements lors des manipulations de l'utilisateur au modèle de présentation (1).
  - Il modifie ses caractéristiques visuelles chaque fois qu'il est averti que le modèle de présentation a été modifié (7).
- Modèle de présentation : son rôle est de conserver l'état du dialogue avec l'utilisateur et de masquer les propriétés des événements liées aux choix de composants graphiques. L'objectif de ce type de classe est d'isoler les logiques métier des interactions avec l'utilisateur qui ne sont pas directement liées à la logique métier comme, par exemple, quel est l'élément (école, formation ou cours) sélectionné ou quel est l'utilisateur actuellement connecté. Le modèle de présentation communique avec les autres classes et :
  - Il soulève un événement métier (2) en réponse à la réception d'un événement d'un composant graphique (1).

- Il prévient le composant graphique de toutes les modifications de son état (7) provoquées par la réception d'un événement de modification du modèle métier (5) ou à cause d'une modification provoquée par le contrôleur (3).

Le contrôleur contient la logique de communication avec l'utilisateur. Il réagit aux événements métier envoyés par le modèle de présentation. Il peut suivant les cas :

- Interroger ou modifier le modèle de présentation (3).
- Provoquer une modification du modèle métier (4).
- En cas de refus de modification par le modèle métier, il lui appartient de gérer la communication avec l'utilisateur en passant par le modèle de présentation.

Le modèle métier contient les données métier et la logique de leur modification et de leur stockage :.

- Les demandes de modification (appel de méthodes) sont envoyées par les contrôleurs (4).
- En cas de réussite, le modèle soulève un événement de modification de propriété prévenant le modèle de présentation (5).
- En cas d'échec, le contrôleur est prévenu soit par la réception d'une exception soit par la valeur de renvoi de la méthode (6).

7.4. Détail des classes de l'application

7.4.1. Interface de school.client.commons

- `Factory` : fabrique générale gérant la création et l'initialisation des instances de l'application.
- `NetFactory` : fabrique abstraite permettant de récupérer une `URL` utilisant le protocole adapté à la récupération de dossiers de cours.
- Gestion des modèles métier
  - `DataFactory` : fabrique abstraite donnant accès aux implémentations de `School`, `Training` et `Lesson`.
  - `Shcool` : école contenant la liste de ses formations.
  - `Training` : formation contenant la liste de ses cours.
  - `Lesson` : cours donnant accès à son dossier de cours.
- Gestion des contrôleurs
  - `ActionFactory` : fabrique abstraite donnant accès aux implémentations concrètes des `ActionModelListener`s.
  - `ActionModelListener` : interface utilisée pour implémenter les contrôleurs de l'application.

- `ActionCommand` : énumération des actions possibles d'un utilisateur.
- `ActionModelEvent` : événement envoyé à la méthode `actionModelePerformed` des `ActionModelListeners`.

- Gestion des curriculums
  - `CurriculumFactory` : fabrique abstraite donnant accès aux curriculums de formation.
  - `Curriculum` : liste de noms de cours

7.4.2. Organisation des fabriques

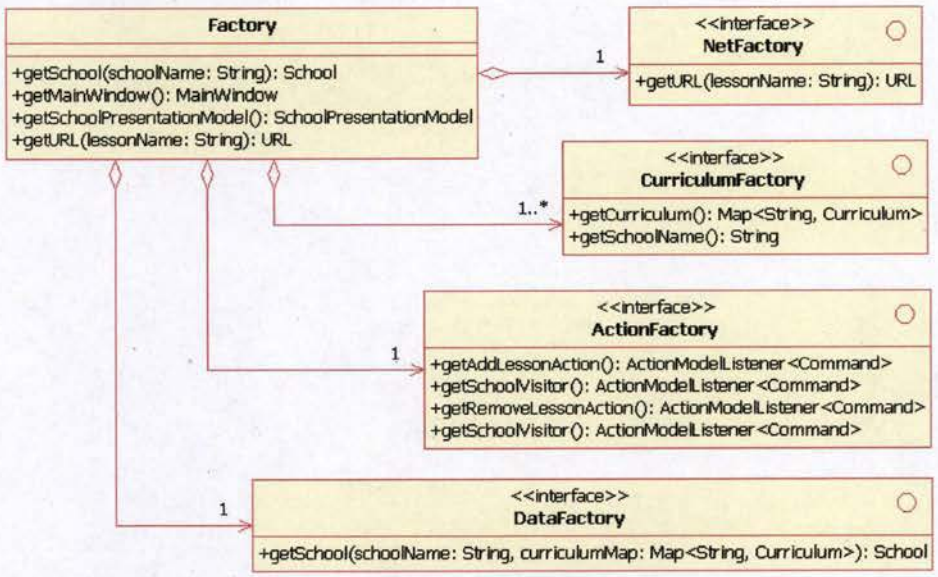

Figure 24 : Diagramme de classes des fabriques

Factory :

- La `NetFactory` définit le protocole utilisé pour accéder au dossier de cours. Dans la version prototype de l'application, elle est initialisée avec une instance de `school.net.NetFactoryImpl`.
- La collection de `CurriculumFactory` définit l'ensemble des partenaires fournisseurs de curriculums présents dans l'application. Dans la version prototype une seule `CurriculumFactory` est présente dans la collection `school.curriculum.CurriculumFactoryImpl`.
- L'`ActionFactory` définit les contrôleurs des actions de l'utilisateur. Dans la version prototype elle est initialisée avec une `school.client.action.ActionFactoryImpl`.
- La `DataFactory` définit l'implémentation des modèles métier utilisés par l'application. Dans la version prototype elle est initialisée avec une instance de `school.client.data.DataFactoryImpl`.

- `getSchool(schoolName :String)` retourne l'instance de l'école, initialisée avec les formations et les cours correspondants aux curriculums fournis par la `CurriculumFactory` et dont la propriété `SchoolName` correspond à la valeur du paramètre.
- `getMainWindow()` retourne la référence de la vue principale de l'application. Cette vue est initialisée avec, comme écouteur des différents événements utilisateurs, l'instance du `SchoolPresentationModel` retourné par la méthode `getSchoolPresentationModel()`.
- `getSchoolPresentationModel()` retourne le modèle de présentation des vues de l'application. Il a comme rôles de :
  - Représenter l'état du dialogue avec l'utilisateur. Toute modification de cet état est automatiquement répercutée sur les éléments visuels de l'application.
  - Abstraire les événements liés aux composants visuels pour en faire des événements métier. Les événements envoyés par les composants visuels sont reçus par le modèle de présentation, transformés en événements modèles `ActionModelEvent` et redirigés vers les écouteurs de celui-ci.

    L'instance retournée par la méthode est initialisée avec comme écouteur de ses propriétés, l'instance renvoyée par `getSchoolMainWindow()` et comme écouteurs des événements métier, les différents contrôleurs définit par `ActionFactory`.
- `getURL(lessonName:String)` retourne une `URL` vers un dossier de cours en fonction du nom de cours passé en paramètre. Dans la version prototype, la fonctionnalité est simplement déléguée au `NetFactory`.

NetFactory :

- `getURL(lessonName:String)` retourne une `URL` vers un dossier de cours en fonction du nom de cours passé en paramètre.

CurriculumFactory

- `getCurriculumMap()` retourne une collection clés-valeurs dont les clés sont les noms des formations et les valeurs les curriculums des formations correspondantes.
- `getSchoolName()` retourne le nom de l'école fournissant les curriculums.

ActionFactory :

- `getAddLessonAction()` retourne le contrôleur gérant l'ajout d'un cours.
- `getSchoolVisitor()` retourne le contrôleur gérant le changement d'éléments (cours, formations, école) actif.
- `getRemoveLessonAction()` retourne le contrôleur gérant la suppression d'un cours d'une formation.

- `getLoginAction()` retourne le contrôleur gérant l'ouverture et la fermeture de session par un utilisateur. La version prototype ne gère que l'aspect graphique de l'action.

DataFactory :

- `getSchool(schoolName, curriculumMap)` retourne une nouvelle école initialisée avec les formations et les cours correspondants aux `curriculumMap` et dont le nom est égal à la valeur du paramètre `schoolName`.

7.4.3. Interfaces communes

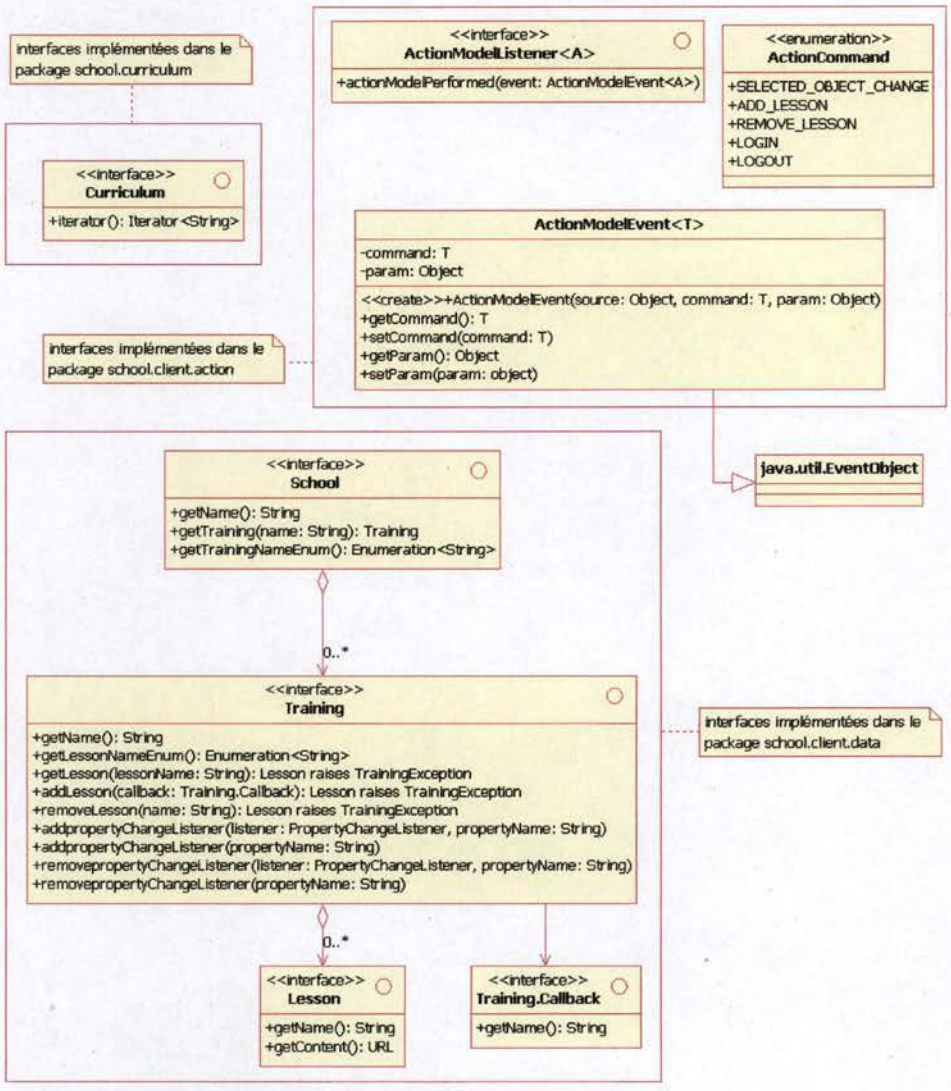

Figure 25 : Diagramme de classes des interfaces communes

Curriculum : liste des noms de cours d'une formation :

- `Iterator()` retourne un nouvel *Iterator* des noms de cours.

ActionModelListener<A> : interface que doivent implémenter les classes dont les instances veulent s'inscrire comme écouteur d'actions métier :

- `A` : type des actions écoutées par la classe.
- `actionModelPerformed(event)` méthode appelée lorsqu'un événement est déclenché par l'instance écoutée.

ActionModelEvent<T> : événement métier défini par une action et un objet paramètre :

- `T` : type des objets représentant l'action. Dans le cadre de cette application, toutes les actions sont représentées par des objets de type `ActionCommand`.
- `command` : action qui a provoqué l'événement.
- `ActionModelEvent(source, command, param)` constructeur initialisant les propriétés de l'instance avec la valeur des paramètres. Le paramètre source est conservé par l'ancêtre `java.util.EventObject`.
- `param` est l'objet paramètre.
- `getCommand()` retourne l'action.
- `setCommand()` modifie l'action.
- `getParam()` retourne le paramètre.
- `setparam()` modifie le paramètre.

ActionCommand : énumération des actions d'un utilisateur dans l'application :

- `SELECTED-OBJECT-CHANGE` : l'utilisateur modifie la sélection de l'objet actif (école, formation ou cours).
- `ADD_LESSON` : l'utilisateur ajoute un cours.
- `REMOVE_LESSON` : l'utilisateur supprime un cours.
- `LOGIN` : l'utilisateur ouvre une session.
- `LOGOUT` : l'utilisateur ferme une session.

School : école définie par un nom et un ensemble de formations.

- `getName()` retourne le nom de l'école.
- `getTraining(trainingName)` retourne la formation dont le nom est égal au nom passé en paramètre.
- `getTrainingNameEnum()` retourne une énumération de noms de formations proposées par l'école.

Training : formation définie par un nom et un ensemble de cours.

- `getName()` retourne le nom de la formation.
- `getLessonNameEnum()` retourne une énumération de noms de cours de la formation.

- `getLesson(lessonName)` retourne le cours correspondant au nom du cours passé en paramètre. La méthode soulève une `TrainingException` si le cours n'existe pas pour cette formation.
- `addLesson(callback)` crée et ajoute un cours à la formation. Le cours est créé à partir du nom retourné par la méthode `getName()` du `Callback` passé en paramètre. Cette technique d'inversion de contrôle permet de ne demander le nom du cours à l'utilisateur que si la formation accepte l'ajout d'un cours. La méthode soulève une `TrainingException` si le nom du cours n'est pas valide ou qu'il est déjà présent dans cette formation.
- `removeLesson(LessonName)` supprime un cours de l'ensemble des cours d'une formation. Le cours supprimé sera celui dont le nom est égal au paramètre. La méthode soulève une `TrainingException` si le cours n'est pas présent.
- `addPropertyChangeListener` et `removePropertyChangeListener` permettent à un `PropertyChangeListener` de s'inscrire ou de se supprimer comme écouteur des événements d'ajout et de suppression de cours. Les propriétés à écouter sont `removeLesson` et `addLesson`.

Training.Callback : interface des classes permettant à la méthode `addLesson()` d'une formation de récupérer le nom du cours à ajouter :

- `getname()` retourne le nom du `Callback`.

Lesson : représente un cours défini par un nom et un dossier de cours :

- `getName()` retourne le nom du cours.
- `getContent()` retourne le dossier de cours sous la forme d'une `Url`.

8. Consignes de travail à destination des étudiants

Pour se rendre compte du niveau de leurs compétences, la société Duke demande aux candidats programmeurs, pour chaque objectif, de fournir une réponse à une série de questions et le code d'implémentation de solutions concrètes.

Le travail à remettre sera constitué :

- Du code modifié du prototype fourni.
- D'un rapport reprenant les réponses aux questions et les modifications de code.

Le rapport sera organisé à partir du format électronique du présent document, complété par les réponses à la suite de chacune des questions.

Les réponses aux questions seront documentées avec des liens vers des sites de référence pertinents.

Les implémentations seront directement ajoutées au code du prototype fourni. Elles seront également copiées et commentées dans le rapport en regard de chaque demande.

8.1. Comprendre les principes de la cryptographie

8.1.1. Questions

1 Expliquer les différences, les avantages et inconvénients des trois fonctions cryptographiques suivantes :

- o Chiffrement symétrique.
- o Chiffrement asymétrique.
- o Fonction de hachage sécurisée.

2 Quels algorithmes sont les plus utilisés dans chacune des fonctions ci-dessus ? Qu'elle est la longueur des clés que vous conseilleriez pour chacune de ces fonctions ?

3 Définir les termes suivants :

- o Clé secrète.
- o Clé privée.
- o Clé publique.
- o Code d'authentification de message.
- o Certificat.

- Autorité de certification.

8.1.2. Implémentation de solutions

Les implémentations de la liste suivante sont à concevoir sous la forme d'un code élémentaire en langage batch.

1 Automatiser l'installation d'une autorité de certification avec *OpenSSL*

2 Automatiser la création d'un certificat personnel permettant la signature de code :

- Génération d'une clé publique et privée avec « keytool ».
- Génération de la requête de certificat avec « keytool ».
- Création du certificat garanti par l'autorité de certification créée au point 1.
- Ajout du certificat à la *keystore*.

8.2. Signature de code

Les garanties de qualité que donne la société Duke ne concernent que le code qu'elle certifie. L'utilisateur doit pouvoir contrôler à tout moment qu'un code étranger à la société Duke n'ait pas accès aux ressources protégées. Pour cela, la société Duke envisage de signer tous les codes qu'elle distribue et de mettre en place une procédure automatique de vérification de l'intégrité du code qui s'exécute.

8.2.1. Questions

1 Définir ce qu'est un signature numérique

8.2.2. Implémentation de solutions

1 Automatiser sous la forme d'un code élémentaire en langage batch la signature des différents fichiers « jar » de l'application prototype.

2 Création d'une tâche dans le fichier « build.xml » pour automatiser la signature des fichiers « jar ».

3 Modifier le fichier « ecole.policy » pour que seul le code des fichiers « jar » signés avec le certificat puisse avoir des permissions. **Remarque** : cette condition sur les permissions doit être vérifiée lors de toute modification du fichier « ecole.policy ».

8.3. Gestion du contrôle d'accès

L'application à développer doit pouvoir contrôler l'accès aux données en fonction de permissions configurables par les administrateurs des sociétés où l'application sera installée.

8.3.1. Questions

1 Quels sont les rôles dans la gestion du contrôle d'accès aux ressources sensibles des classes suivantes :

- `java.lang.ClassLoader`
- `java.lang.SecurityManager`
- `java.security.AccessController`
- `java.security.AccessControllerContext`
- `java.security.Policy`

2 Définir rapidement ce qu'est un domaine de protection et ses différentes propriétés :

- `java.security.ProtectionDomain`
- `java.security.CodeSource`
- `java.security.Permissions`
- `java.security.Principal`

3 Expliquer le fonctionnement dans les classes suivantes de la méthode `checkPermission (Permission)`:

- `java.lang.SecurityManager`
- `java.security.AccessController`
- `java.security.AccessControllerContext`

4 Expliquer le fonctionnement dans les classes suivantes de la méthode `implies(Permission)` :

- `java.security.Permission` et ses descendants
- `java.security.ProtectionDomain`
- `java.security.Permissions`
- `java.security.PermissionCollection`

5 Expliquer le fonctionnement des méthodes suivantes de la classe `java.security.AccessController`:

- ```
  public static <T> T doPrivileged
   (PrivilegedExceptionAction<T> action)
   throws PrivilegedActionException
  ```
- ```
  public static <T> T doPrivileged
   (PrivilegedAction<T> action)
  ```

8.3.2. Implémentation de solutions

1 Créer une tâche « Ant » dans le fichier « `build.xml` » permettant de démarrer l'application cliente sous le contrôle du *SecurityManager* par défaut. L'application utilisera :

- sun.security.provider.PolicyFile comme implémentation de *java.security.Policy*
- exclusivement un fichier « `ecole.policy` » créé à partir d'une copie de « `<JAVA_HOME>/lib/security.java.policy` » pour la définition des permissions.

2 Modifier le fichier « `ecole.policy` » en donnant les permissions minimales en fonction du *codeBase* correspondant aux différents fichiers « jar » de déploiement.

3 Créer des contextes privilégiés de façon à, sans compromettre la sécurité, limiter au maximum le nombre de fichiers « jar » ayant besoin d'une permission d'accès aux ressources.

4 Modifier le code pour que l'accès à une formation soit vérifiée par le *SecurityManager*. Créer pour cela un nouveau type de permission `school.client.data.TrainingAccesPermission`. Le nom de la permission aura comme syntaxe :

- *TrainingAccessPermissionName* :

  `ecoleSecur`. *SchoolDefinition*

- *SchoolDefinition* :

  *

  | *SchoolName* . *TrainingDefinition*

- *SchoolName* : la valeur nom de l'école concernée.
- *TrainingDefinition* :

  *

  | *TrainingName*

- *TrainingName* : la valeur nom de la formation concernée.

5 Modifier le fichier « `ecole.policy` » pour ne donner l'accès qu'à la formation « java » de l'école « STE-Formations ».

6 Sur le même modèle que les `TrainingAccesPermission`, créer des `school.client.data.LessonAccesPermission` pour contrôler l'accès aux cours d'une formation. Le nom de la permission aura comme syntaxe :

- *LessonAccessPermissionName* :

  `ecoleSecur`. *SchoolDefinition*

- *SchoolDefinition* :

  *

  | *SchoolName* . *TrainingDefinition*

- *SchoolName* : la valeur nom de l'école concernée.
- *TrainingDefinition* :

  *

  | *TrainingName* . *LessonDefinition*
- *TrainingName* : la valeur nom de la formation concernée.
- *LessonDefinition* :

  *

  | *LessonName*
- *LessonName* : la valeur nom de la formation concernée.

7 Modifier le fichier « ecole.policy » pour ne donner l'accès qu'au cours de la formation « java » de l'école « STE-Formations ».

8 Modifier le code pour que l'ajout et la suppression d'un cours à une formation soient vérifiés par le *SecurityManager*. Créer pour cela une classe `school.client.data.TrainingUpdatePermission` qui sera définie par un nom et une ou des actions. L'implémentation utilisera l'interface `java.security.Guard` pour contrôler les actions.

- Les noms des permissions ont la même syntaxe que les `TrainingAccessPermissions` et sont gérés comme les noms dans les `BasicPermission`.
- Les actions possibles sont :
  - « `all` » : représente l'ajout et la suppression.
  - « `add` » : représentant l'ajout.
  - « `delete` » :représentant la suppression.
- Les détails de l'implémentation de la classe sont décrits par une série de tests unitaires contenus dans la classe de *JUnitTest* `school.client.data.TestTrainingUpdatePermission`.

8.4. Sécurisation de la communication réseau

Pour garantir la confidentialité et l'authenticité des dossiers de formation présentés par son application, la société Duke envisage d'utiliser l'implémentation fournie par le service « *Java™ Secure Socket Extension* » (JSSE).

8.4.1. Questions

1 Citer les différentes options de paramétrage d'une connexion sécurisée avec TLS/SSL.

2 Expliquer le rôle de chacune des techniques de chiffrement (symétrique, asymétrique, fonction de hachage sécurisée) dans une communication sécurisée de type TLS/SSL.

3 Expliquer à quoi sont utilisés les éléments suivants dans une communication sécurisée TLS/SSL entre un client et un serveur :

- Le certificat de l'autorité de certification.
- La clé privée de l'autorité de certification.
- La clé publique de l'autorité de certification.
- Le certificat du serveur.
- La clé privée du serveur.
- La clé publique du serveur.
- Le certificat du client.
- La clé privée du client.
- La clé publique du client.

8.4.2. Implémentation de solutions

Concevoir un code élémentaire en langage batch permettant :

1 La création et l'installation des certificats dans les *keystores* client et serveur :

- Génération d'une clé publique et privée.
- Génération de la requête de certificat.
- Création du certificat garanti par l'autorité de certification.
- Ajout des certificats de l'autorité et du serveur à la *keystore* serveur.
- Ajout du certificat de l'autorité de certification à la *trusted keystore* cliente.

2 Créer une tâche « Ant » dans le fichier « `build.xml` » du prototype permettant de démarrer le serveur avec la configuration nécessaire pour une connexion SSL avec authentification du serveur.

3 Créer une tâche « Ant » dans le fichier « `build.xml` » permettant de démarrer le client avec la configuration nécessaire pour une connexion SSL avec authentification du serveur.

4 Modifier le code client et le code serveur pour qu'ils utilisent des sockets sécurisés.

8.5. Authentification et autorisation

Les sociétés où sont installées l'application possèdent, pour la plupart, un système d'authentification des utilisateurs de leur parc machine. La gestion de l'inscription des utilisateurs et de leurs identifiants de sécurité est un travail important. L'application à développer devra pouvoir utiliser les procédures internes des sociétés pour ne pas obliger la gestion en doublon des utilisateurs. La société Duke envisage d'utiliser le service JAAS (*Java™ Authentication and Authorization Service*) pour atteindre cet objectif.

8.5.1. Questions

1 Définir ce que représentent les classes et interfaces suivantes ainsi que leurs relations :

- `javax.security.auth.Subject`
- `java.security.Principal`
- `javax.security.auth.login.LoginContext`
- `javax.security.auth.spi.LoginModule`
- `javax.security.auth.callback.CallbackHandler`
- `javax.security.auth.callback.Callback`

2 Détailler la logique d'implémentation de la méthode `handle(Callback[])` des `CallbackHandler`.

3 Citer, définir et donner la valeur par défaut des clés de configuration utilisées par le service JAAS.

4 Expliquer ce que font les méthodes suivantes de la classe `javax.security.auth.Subject` :

- ```
  public static <T> T doAs( final Subject subject,
   final java.security.PrivilegedAction<T> action)
  ```
- ```
  public static <T> T doAs( final Subject subject,
   final java.security.PrivilegedExceptionAction<T> action)
   throws java.security.PrivilegedActionException
  ```
- ```
  public static <T> T doAsPrivileged(
   final Subject subject,
   final java.security.PrivilegedAction<T> action,
   final java.security.AccessControlContext acc)
  ```
- ```
  public static <T> T doAsPrivileged(
   final Subject subject,
   final java.security.PrivilegedExceptionAction<T> action,
   final java.security.AccessControlContext acc)
   throws java.security.PrivilegedActionException
  ```

8.5.2. Implémentation de solutions

1 Configurer l'application pour qu'elle puisse démarrer en utilisant le `LoginModule` de l'exemple proposé par SUN.

- Ajouter à l'application prototype les classes de l'exemple du tutorial de la version 1.4.2 « JAAS Authentication Tutorial » :
  - `sample.module.SampleLoginModule` (http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/tutorials/SampleLoginModule.java).
  - `sample.module.SamplePrincipal` (http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/tutorials/SamplePrincipal.java).
- Créer un fichier de configuration permettant de charger ce module.
- Créer une tache « Ant » permettant de lancer l'application avec les paramètres nécessaires à l'utilisation du fichier de configuration.

2 Modifier l'application pour que son interface de connexion utilise le `LoginModule` à travers le `LoginContext` initialisé à l'étape 1.

3 Modifier l'application pour que la vérification d'accès aux ressources protégées se fasse en tenant compte de l'utilisateur connecté.

4 Modifier le fichier « ecole.policy » pour que les permissions soient accordées en fonction de l'utilisateur connecté.

Conclusion

Le domaine de la sécurité en programmation est particulièrement vaste et complexe.

La programmation, avec les contraintes de sécurité, complique fortement l'implémentation des fonctionnalités car, non seulement, il faut qu'elles produisent le résultat attendu mais, en plus, elles ne doivent produire que celui-là. Elles doivent l'interdire si certaines conditions extérieures ne sont pas rencontrées.

Nous avons montré dans ce mémoire que l'on pouvait aborder cette problématique avec de jeunes programmeurs qui ont comme préoccupation prioritaire de produire un code qui fonctionne.

Nous avons montré qu'une démarche basée sur une mise en situation proche de leur futur contexte professionnel permettait aux étudiants de réaliser un processus d'apprentissage et de recherche personnelle sur une matière complexe.

Ils ont pu expérimenter concrètement l'implémentation de sécurité dans des domaines très demandés dans le monde de l'entreprise.

L'introduction des concepts de base en les détachant de tout langage ainsi qu'une partie pratique consistant à la modification d'une application prototype ont permis de renforcer des apprentissages techniques transversaux communs à l'entièreté de la formation comme :

- Bonne logique de programmation.
- Bonne connaissance du langage Java.
- Bonne compréhension des concepts de la programmation objet.
- Connaissance des protocoles réseaux.
- Connaissance des Design patterns les plus courants (dont MVC).

Les échanges entre les étudiants ont favorisé une auto-construction de leurs savoirs et leur ont donné confiance dans leurs capacités à surmonter des difficultés qui à première vue paraissent inabordables.

La rédaction d'un rapport collectif amène les étudiants à formuler et à confronter leur apprentissage ce qui permet tout à la fois une évaluation permanente de l'étudiant sur son niveau de compréhension de la matière mais aussi de renforcer l'acquisition de celle-ci.

Lors de la rédaction et de la mise en commun, l'étudiant est confronté à sa capacité à reformuler ce qu'il a appris et à se poser la question : cela correspond-il à ce que les autres ont compris de la matière ?

La reformulation d'un point de matière oblige à structurer son savoir et à préciser les matières encore vagues.

Grâce à la rédaction de ce mémoire, nous avons pu réaliser un travail de fond sur la préparation de ce cours et ainsi le faire évoluer, pour la plus grande satisfaction des étudiants et de la nôtre. Nous sommes conscients que ce travail est perfectible. Nous dégageons des pistes d'évolution dans le chapitre suivant.

Expérimentation et évaluation

Le cours présenté dans ce mémoire, a été donné une première fois. Il a été évalué par les étudiants de façon informelle lors d'un suivi de groupe, réunion durant laquelle les étudiants expriment leurs éventuelles difficultés et donnent un feed-back sur les différents cours. Il semble que la méthodologie employée dans le cours leur donne satisfaction et ils considèrent que les objectifs (au moins techniques) sont atteints. En fin de formation, les étudiants réalisent un travail pratique, reprenant les matières fondamentales vues en formation, dont la sécurité. Il apparaît que globalement, ils n'ont pas éprouvé de difficultés pour l'implémentation des techniques de sécurité. Les documents réalisés dans le cadre du cours leur ont été très utiles.

Par ailleurs, les travaux de synthèse demandés aux étudiants dans le cadre du cours, qui nous le rappelons, se font en ligne via un wiki, et que nous pouvons visualiser donc en temps réel, nous permettent une évaluation immédiate de leur progression et des difficultés rencontrées. Nous pouvons ainsi mieux cerner mon public et améliorer notre réactivité.

Nous regrettons cependant de ne pas avoir eu le temps de leur proposer une évaluation formelle du cours qui nous aurait permis d'analyser plus finement l'impact de celui-ci. Ce sera fait pour la prochaine session.

En tant que formateur, nous avons éprouvé un réel plaisir à expérimenter cette méthodologie, même si elle n'est pas de tout repos. Elle nécessite de la part du formateur beaucoup de vigilance, d'implication personnelle et une organisation sans faille.

Nous devons avouer également que cette expérience a été menée sur un groupe d'étudiants particulièrement motivés, très collaboratifs et techniquement doués. Ce ne sera très certainement pas toujours le cas. Il faut nous y préparer.

Evolutions techniques

Le cours n'envisage pas tous les aspects de sécurité. Nous relèvons deux manquements majeurs :

1 Le premier concerne les bonnes pratiques pour coder "sécurisé". Les sites de référence, qui abordent cette problématique, donnent des conseils qui dépassent largement le cadre de sécurité et qui sont plutôt des bonnes pratiques en matière de programmation servant les intérêts de la sécurité. Par exemple, il est recommandé de choisir des identificateurs de variables visuellement très différents afin d'éviter toute confusion de la part du programmeur. Si dans tous nos cours de programmation, nous veillons aux bonnes pratiques en matière d'encapsulation, d'héritage, d'usage de

Designs Pattern, etc., nous n'insistons pas assez sur les bonnes pratiques dont l'objectif est de soutenir la sécurité.

2 Le deuxième concerne toute la sécurité relative aux conteneurs d'objets , dans un contexte de serveurs comme Tomcat, EJB. Dans ce type d'environnement, le contrôle d'accès et la gestion des sessions des utilisateurs ne se situent pas dans le code mais au niveau des objets générés par le serveur qui encapsulent les objets de l'application. La sécurité se gère au niveau de la configuration des serveurs et non dans le code de l'application. L'intégration de cette technique demande préalablement une connaissance de la programmation de ce type d'environnement. Le temps imparti au cours de sécurité n'autorise malheureusement pas cette approche actuellement.

Bibliographie

- Benoît Charroux, Aomar Osmani, Yann Thierry-Mieg, *UML 2 : Pratique de la modélisation*, 2ème édition, Pearson Education, France, 2008
- David Flanagan, *Java en concentré : Manuel de référence*, 5e édition, O'Reilly, Paris, 2006
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns : Catalogue de modèles de conception réutilisables*, Vuibert Informatique, Paris, 1999
- Li Gong, Gary Ellison and Mary Dageforde, *Inside Java 2 Platform Security, Second Edition: Architecture, API Design and Implementation*, Addison-Wesley, Boston, Mai 2003
- Malcom Knowles, *L'adulte apprenant*, Organisations, 1990
- Edward Lindeman, *The meaning of adult education*, 1926
- Scott Oaks, *Sécurité en Java*, O'Reilly, Paris, 1999

Request for comments (RFC)

- T. Dierks Independent, E. Rescorla RTFM, Inc, The Transport Layer Security (TLS) Protocol Version 1.2, Network Working Group, Request for Comments : RFC 5246 , http://www.rfc-archive.org/getrfc.php?rfc=5246 (Août 2008) (Date de consultation : Juillet 2009)
- H. Krawczyk IBM, M. Bellare UCSD, R. Canetti IBM, HMAC : Keyed-Hashing for Message Authentication, Network Working Group Request for Comments : RFC 2104, http://www.rfc-archive.org/getrfc.php?rfc=2104 (Février 1997) (Date de consultation : Juillet 2009)
- E. Rescorla, Diffie-Hellman Key Agreement Method : RFC 2631, http://www.rfc-archive.org/getrfc.php?rfc=2631 (Juin 1999) (Date de consultation : Juillet 2009)
- R. Rivest, The MD5 Message-Digest Algorithm : RFC 1321, http://www.rfc-archive.org/getrfc.php?rfc=1321 (Avril 1992) (Date de consultation : Juillet 2009)

Sites de références

- Direction centrale de la sécurité des systèmes d'information, Secrétariat général de la défense nationale, Mécanismes cryptographiques, http://www.ssi.gouv.fr/site_documents/politiqueproduit/Mecanismes_cryptographique_v1_10_standard.pdf (Version 1.10, 2006) (Date de consultation : Juillet 2009)
- Li Gong, JavaTM SE Platform Security Architecture, http://java.sun.com/javase/6/docs/technotes/guides/security/spec/security-spec.doc.html (Version 1.2, 1999-2002) (Date de consultation : Juillet 2009)

- Sun Microsystems, Inc, Default Policy Implementation and Policy File Syntax, : http://java.sun.com/javase/6/docs/technotes/guides/security/PolicyFiles.html (Revision 1.6 2002) (Date de consultation : Juillet 2009)
- Sun Microsystems, Inc, JavaTM Authentication and Authorization Service (JAAS), Reference Guide, for the Java TM SE Development Kit 6 : http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html (Date de consultation : Juillet 2009)
- Sun Microsystems, Inc, Java™ Secure Socket Extension (JSSE), Reference Guide for Java™ Platform Standard Edition 6, http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html, 1999-2006, (Date de consultation : Juillet 2009)

A.I. Illustration de la technique Diffie-Helleman

Feuille de calcul Excel illustrant la technique Diffie-Helleman de détermination de clé secrète.

Algorithme de Diffie-Hellman

Alice		espion	Bernard	
x	15		y	13
g	47		g	47
n	127		n	127
g^x mod n	4	127 47 4	g^x mod n	4
g^y mod n	122	122	g^y mod n	122
((g^y mod n)^x) mod n	**32**		**((g^x mod n)^y)mod n**	**32**

Cet algorithme est très simple pour l'échange des clefs et le voici décrit :

Soit 2 personnes A et B désirant communiquer sans utiliser une clé secrète.

Pour cela elles se mettent d'accord sur 2 nombres **g** et **n** tels que n soit supérieur à g et g supérieur à 1, et cela sur un canal non sécurisé (il faut que n soit grand, de l'ordre de 1024 bits pour que l'échange des clés soit sécurisé).

En plus, elles choisissent chacune un nombre aléatoire secret **x** et **y**:

- A choisit **x** ,calcule **X=g^x mod n** et l'envoie à B.

- B choisit **y** ,calcule **Y=g^y mod n** et l'envoie à A.

Ainsi le pirate peut intercepter **g** et **n** mais il lui est impossible d'en déduire x et y (c'est sur ce principe que repose la sécurité de l'algorithme).

De son côté, A calcule **k=Y^x mod n** et B calcule **k'=X^y mod n**.

Si l'on regarde de plus près, on voit quelque chose de très intéressant : **k=k'=g^xy mod n**. Ainsi, A et B ont réussi à créer une clé privée dont ils sont les seuls détenteurs.

Le pirate ,malgré la vision qu'il a eu de l'échange, ne peut calculer la clé privée.

L'échange des clés ayant fonctionné, A et B peuvent communiquer par un algorithme à clé privée.

```
Function modexp(nbr, exp, div)
  If exp < 2 Then
    modexp = nbr Mod div
  Else
    reste = modexp(nbr, exp \ 2, div)
    reste = (reste * reste) Mod div
    If exp Mod 2 = 1 Then
      reste = (reste * nbr) Mod div
    End If
    modexp = reste
  End If
End Function
```

Excel transforme automatiquement les grandes valeurs entières en une approximation exprimée en valeurs flottantes. Ce type de comportement n'étant pas du tout approprié à la détermination d'une clé de chiffrement, la feuille de calcul est accompagné du code d'une fonction `modexp(nbr, exp, div)` permettant itérativement de calculer nbr^{exp} `modulo div` tout en restant dans les limites d'un entier.

A.II. Code de l'application prototype

school.client.commons.Factory

```
package school.client.commons;

import java.net.MalformedURLException;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;

import school.client.action.ActionFactoryImpl;
import school.client.data.DataFactoryImpl;
import school.client.gui.MainWindow;
import school.client.gui.SchoolPresentationModel;
import school.curriculum.CurriculumFactoryImpl;
import school.net.NetFactoryImpl;

/**
 * Fabrique générale gérant la création et l'initialisation des instances de
 * l'application.
 *
 * @author boogaerts
 *
 */
public class Factory {

 private static Factory instance;
 /**
  * Définit le protocole utilisé pour accéder au dossier de cours. Dans la
  * version prototype de l'application elle est initialisé avec une instance
  * de {@link NetFactoryImpl}.
  */
```

```
private final NetFactory supportFactory;
/**
 * Définit l'implémentation des modèles métiers utilisés par l'application.
 * Dans la version prototype elle est initialisée avec une instance de
 * {@link DataFactoryImpl}.
 */
private final DataFactory dataFactory;
/**
 * Définit les contrôleurs des actions de l'utilisateur. Dans la version
 * prototype elle est initialisé avec une {@link ActionFactoryImpl}.
 */
private final ActionFactory userFactory;
/**
 * Définit l'ensemble des partenaires fournisseurs de curriculums présents
 * dans l'application. Dans la version prototype une seule CurriculumFactory
 * est présente dans la collection : {@link CurriculumFactoryImpl}.
 */
private Map<String, CurriculumFactory> contentsfactoryMap;
private MainWindow mainWindow;
private SchoolPresentationModel schoolPresentationModel;

private Factory() {
 super();
 this.dataFactory = DataFactoryImpl.getInstance();
 this.userFactory = new ActionFactoryImpl();
 this.supportFactory = new NetFactoryImpl();
 this.contentsfactoryMap = getCurriculumFactoryMap();
}
private Map<String, CurriculumFactory> getCurriculumFactoryMap() {
 if (this.contentsfactoryMap == null) {
  this.contentsfactoryMap = new HashMap<String, CurriculumFactory>();
  CurriculumFactory curriculumFactory = new CurriculumFactoryImpl();
  this.contentsfactoryMap.put(curriculumFactory.getSchoolName(),
    curriculumFactory);
 }
 return this.contentsfactoryMap;
}
public static Factory getInstance() {
 if (Factory.instance == null) {
  Factory.instance = new Factory();
 }
 return Factory.instance;
}
/**
 * Retourne l'instance de l'école initialisée avec les formations et les
 * cours correspondants aux curriculums fournis par la CurriculumFactory
 * dont la propriété SchoolName correspond à la valeur du paramètre.
 *
 * @param schoolName
 *      le nom de l'école.
 * @return l'école.
 */
public School getSchool(String schoolName) {
 School school;
 CurriculumFactory curriculumFactory = this.getCurriculumFactoryMap()
   .get(schoolName);
 if (curriculumFactory != null) {
  school = this.dataFactory.getSchool(schoolName, curriculumFactory
    .getCurriculumMap());
 } else {
  school = this.dataFactory.getSchool(schoolName,
    new HashMap<String, Curriculum>());

 }
 return school;
}
/**
 * Retourne la référence de la vue principale de l'application. Cette vue
 * est initialisée avec comme écouteur des différents événements
 * utilisateurs l'instance du {@link SchoolPresentationModel} retourné par
 * la méthode {@link #getSchoolPresentationModel()}.
 *
 * @return la vue principale de l'application.
 */
public MainWindow getMainWindow() {
```

```
 if (mainWindow == null) {
  mainWindow = new MainWindow();
  this.mainWindow
    .addTreeSelectionListener(getSchoolPresentationModel());
  this.mainWindow.addJButtonAddListener(getSchoolPresentationModel());
  this.mainWindow
    .addJButtonRemoveListener(getSchoolPresentationModel());
  this.mainWindow
    .addJButtonLoginListener(getSchoolPresentationModel());
 }
 return mainWindow;
}
/**
 * Retourne le persentationModel des vues de l'application. Il a comme rôles
 * de :
 * <ul>
 * <li>Représenter l'état du dialogue avec l'utilisateur. Toute modification
 * de cet état est automatiquement répercuté sur les éléments visuelles de
 * l'application.</li>
 * <li>Abstraire les événements liés aux composants visuels pour en faire
 * des événements métiers. Les événements envoyés par les composants visuels
 * sont reçu par le « PresentationModel » et transformé en événement modèle
 * ActionModelEvent et ré envoyé aux écouteurs de celui-ci.</li>
 * </ul>
 * L'instance retourné par la méthode est initialisée avec comme écouteur de
 * ses propriétés l'instance renvoyée par getSchoolMainWindow() et comme
 * écouteurs des événements métiers les différents contrôleurs définit par
 * la ActionFactory.
 *
 * @return le SchoolPresentationModel.
 */
public SchoolPresentationModel getSchoolPresentationModel() {
 if (this.schoolPresentationModel == null) {
  this.schoolPresentationModel = new SchoolPresentationModel();
  this.schoolPresentationModel.addPropertyChangeListener(this
    .getMainWindow());
  this.schoolPresentationModel.setSchool(this
    .getSchool("STE-Formations"));
  // TODO refactoring mise en place du curriculum
  this.schoolPresentationModel.addActionModelListener(
    ActionCommand.SELECTED_OBJECT_CHANGE, this
      .getSchoolVisitor());
  this.schoolPresentationModel.addActionModelListener(
    ActionCommand.ADD_LESSON, this.getAddLessonAction());
  this.schoolPresentationModel.addActionModelListener(
    ActionCommand.REMOVE_LESSON, this.getRemoveLessonAction());
  this.schoolPresentationModel.addActionModelListener(
    ActionCommand.LOGIN, this.getLoginAction());
  this.schoolPresentationModel.addActionModelListener(
    ActionCommand.LOGOUT, this.getLoginAction());
 }
 return this.schoolPresentationModel;
}
private ActionModelListener<ActionCommand> getSchoolVisitor() {
 return this.userFactory.getSchoolVisitor();
}
private ActionModelListener<ActionCommand> getAddLessonAction() {
 return this.userFactory.getAddLessonAction();
}
private ActionModelListener<ActionCommand> getRemoveLessonAction() {
 return this.userFactory.getRemoveLessonAction();
}
private ActionModelListener<ActionCommand> getLoginAction() {
 return this.userFactory.getLoginAction();
}
/**
 * Retourne une {@link URL} vers un dossier de cours en fonction du nom de
 * cours passé en paramètre. Dans la version prototype la fonctionnalité est
 * simplement déléguée au NetFactory.
 *
 * @param lessonName
 *      le nom du cours
 * @return l'URL du dossier
 * @throws MalformedURLException
 */
```

```
 public URL getURL(String lessonName) throws MalformedURLException {
  return this.supportFactory.getURL(lessonName);
 }
}
```

school.client.commons.ActionFactory

```
package school.client.commons;

/**
 * Fabrique abstraite donnant accès aux Implémentations concrètes des
 * {@link ActionModelListener}
 *
 * @author boogaerts
 *
 */
public interface ActionFactory {

 /**
  * Retourne le contrôleur gérant l'ajout d'un cours.
  *
  * @return le {@link ActionModelListener} gérant l'ajout d'un cours
  */
 ActionModelListener<ActionCommand> getAddLessonAction();
 /**
  * Retourne le contrôleur gérant le changement d'éléments ( cours
  * {@link Lesson}, formations {@link Training}, école {@link School}) actif.
  *
  * @return le {@link ActionModelListener} gérant le changement d'éléments
  *     actifs
  */
 ActionModelListener<ActionCommand> getSchoolVisitor();
 /**
  * Retourne le contrôleur gérant la suppression d'un cours d'une formation.
  *
  * @return le {@link ActionModelListener} gérant la suppression d'un cours
  */
 ActionModelListener<ActionCommand> getRemoveLessonAction();
 /**
  * Retourne le contrôleur gérant l'ouverture et la fermeture de session par
  * un utilisateur.
  *
  * @return le {@link ActionModelListener} gérant l'ouverture et la fermeture
  *     de session.
  */
 ActionModelListener<ActionCommand> getLoginAction();
}
```

school.client.commons.ActionCommand

```
package school.client.commons;

public enum ActionCommand {
 SELECTED_OBJECT_CHANGE, ADD_LESSON, REMOVE_LESSON, LOGIN, LOGOUT
}
```

school.client.commons.ActionModelListener

```
package school.client.commons;

public interface ActionModelListener<A> {

 public void actionModelPerformed(ActionModelEvent<A> event);
}
```

school.client.commons.ActionModelEvent<T>

```
package school.client.commons;

import java.util.EventObject;

public class ActionModelEvent<T> extends EventObject {

 private static final long serialVersionUID = 1L;
 private T command;
 private Object param;
```

```
public ActionModelEvent(Object source, T command, Object param) {
 super(source);
 this.command = command;
 this.param = param;
}
public T getCommand() {
 return command;
}
public void setCommand(T command) {
 this.command = command;
}
public Object getParam() {
 return param;
}
public void setParam(Object param) {
 this.param = param;
}
@Override
public String toString() {
 return "[source: "+this.getSource()+", command: "+this.getCommand()+
   ", param: "+this.getParam()+"]";
}
}
```

school.client.commons.CurriculumFactory

```
package school.client.commons;

import java.util.Map;

/**
 * Fabrique abstraite donnant accès aux {@link Curriculum}s de formation.
 *
 * @author boogaerts
 *
 */
public interface CurriculumFactory {

 /**
  * Retourne une collection clés valeurs dont les clés sont les noms de
  * formations et les valeurs le curriculum de la formation.
  *
  * @return les Curriculums de formation
  */
 Map<String, Curriculum> getCurriculumMap();
 /**
  * retourne le nom de l'école fournissant les {@link Curriculum}s.
  *
  * @return le nom de l'école.
  */
 String getSchoolName();
}
```

school.client.commons.Curriculum

```
package school.client.commons;

public interface Curriculum extends Iterable<String>{

}
```

school.client.commons.DataFactory

```
package school.client.commons;

import java.util.Map;

/**
 * Fabrique abstraite donnant accès aux implémentations de {@link School},
 * {@link Training} et {@link Lesson}.
 *
 * @author boogaerts
 *
 */
public interface DataFactory {

 /**
```

```
     * Retourne une nouvelle école initialisée avec les formations et les cours
     * correspondants aux curriculumMap et le nom est à la valeur du paramètre
     * schoolName.
     *
     * @param schoolName
     *      le nom de l'école.
     * @param curriculumMap
     *      la définitiondes curriculum par formations.
     * @return la nouvelle école.
     */
    School getSchool(String schoolName, Map<String, Curriculum> curriculumMap);

}
```

school.client.commons.School

```
package school.client.commons;

import java.util.Enumeration;

/**
 * une Ecole est définie par un nom et un ensemble de formations.
 *
 * <pre>
 * Propriétés
 * - nom :String le nom de l'école
 * - formations : Formation[*] l'ensemble des formations dispensés par l'école
 *
 * Contraintes sur les Propriétés
 * - nom ne peut pas être modifié après sa déclaration
 * - nom != null
 * - nom != &amp;quot;&amp;quot;
 * - formations[i].nom = formations[k].nom &amp;lt;=&amp;gt; i = k, avec 0 &amp;lt;= i,k &amp;lt;
formations.lenght
 * </pre>
 *
 * @author Yannick Boogaerts pour STE-Formations
 * @see Training.commun.Formation
 */
public interface School {
    /**
     * Renvoit le nom de l'école.
     *
     * @return la valeur de la propriété nom
     */
    public String getName();
    /**
     * Renvoit la formation dont le nom est égal au paramètre.
     *
     * @param nom
     *      le nom de la formation à récupérer.
     *
     * @return une formation avec formation.nom = nom.
     * @throws TrainingException
     *       si nom ne correspond à aucune formation.
     */
    public Training getTraining(String trainingName) throws TrainingException;
    /**
     * Renvoit une Enumeration des noms des formations de l'école.
     *
     * @return une Enumeration des noms des formations de l'école
     */
    public Enumeration<String> getTrainingNameEnum();
}
```

school.client.commons.Training

```
package school.client.commons;

import java.beans.PropertyChangeListener;
import java.util.Enumeration;

/**
 * Formation définie par un nom et un ensemble de cours.
 *
 * <pre>
```

```
 *  Propriétés
 *  - nom : String //le nom de la formation.
 *  - curriculum : Cours[*] // ensemble des cours de la formation.
 *
 *  Contraintes sur les Propriétés
 *  - nom != null
 *  - nom != &quot;&quot;
 *  - curriculum[i].nom = curriculum[j].nom &lt;=&gt; i = j, avec 0 &lt;= i,j &lt;
curriculum.lenght
 * </pre>
 *
 * @author Yannick Boogaerts pour STE-Formations <br>
 * @see Lesson.commun.Cours
 */
public interface Training {
 /**
  * renvoit le nom de la formation.
  *
  * @return le nom de la formation.
  */
 public String getName();
 /**
  * Renvoit une Enumeration des noms de cours présents dans le curriculum de
  * la formations.
  *
  * @return une Enumeration de noms de cours. Enumeration[i]:String.
  */
 public Enumeration<String> getLessonNameEnum();
 /**
  * Renvoit la référence du cours dont la propriété nom est égals au
  * paramètre. Soulève une FormationException s'il n'y a pas de cours
  * correspondant au paramètre.
  *
  * @param lessonName
  *      le nom du cours demandé.
  * @return la référence du cours dont
  *     curriculum[i].intiutlé.equals(intitule)
  * @throws TrainingException
  *       s'il n'existe pas de curriculum[i].intiutlé.equals(intitule)
  */
 public Lesson getLesson(String lessonName) throws TrainingException;
 /**
  * Crée et ajoute un cours à la formation. Le cours est créé à partir du nom
  * retourné par la méthode getName() du Callback passé en paramètre. Cette
  * technique d'inversion de contrôle permet de ne demander le nom du cours à
  * l'utilisateur que si la formation accepte l'ajout d'un cours. La méthode
  * soulève une TrainingException si le nom du cours n'est pas valide ou
  * qu'il est déjà présent dans cette formation.
  *
  * @param callback
  *      le Callback permettant de récupérer le nom du cours à ajouter.
  * @return le cours ajouté dans le curriculum.
  * @throws TrainingException
  *       si le nom n'est pas valide ou que le nom du cours est déjà
  *       présent dans la formation
  */
 public Lesson addLesson(Training.Callback callback)
   throws TrainingException;
 /**
  * Retire le cours dont le nom est égal au paramètre du curriculum de la
  * formation. Pour que le cours soit retirer il faut qu'il exsite un cours
  * dont le nom est égal au paramètre.
  *
  * @param name
  *      le nom du cours à retirer
  * @return le cours retirer ou null si le cours ne fait pas partie du
  *     curriculum.
  * @throws TrainingException
  */
 public Lesson removeLesson(String name) throws TrainingException;
 public void addPropertyChangeListener(PropertyChangeListener listener);
 public void addPropertyChangeListener(String propertyName,
   PropertyChangeListener listener);
 public void removePropertyChangeListener(PropertyChangeListener listener);
 public void removePropertyChangeListener(String propertyName,
```

```
    PropertyChangeListener listener);
  /**
   * interface des classes permettant à la méthode addLesson() d'une formation
   * de récupérer le nom du cours à ajouter.
   *
   * @author boogaerts
   *
   */
  public interface Callback {
    /**
     * Retourne le nom du Callback
     *
     * @return un nom.
    .*/
    public String getName();
  }
}
```

school.client.commons.Lesson

```
package school.client.commons;

import java.net.URL;

/**
 * un Cours est définit par un nom et un contenu.
 *
 * <pre>
 * Propriétés
 * - nom :String // le nom du cours
 * - contenu :URL // le contenu du cours
 *
 * Contraintes sur les Propriétés
 * Le contenu et l'intitulé ne peuvent pas être null.
 * </pre>
 *
 * @author Yannick Boogaerts pour STE-Formations
 */
public interface Lesson {
    /**
     * Renvoit le nom du cours.
     *
     * @return le nom du cours.
     */
    public String getName();

    /**
     * Renvoit le dossier de cours.
     *
     * @return le dossier de cours.
     */
    public URL getContent();
}
```

school.client.commons.TrainingException

```
package school.client.commons;

/**
 * Exception soulevée pour signaler une action impossible dans la manipulation
 * des formations
 *
 * @author Yannick Boogaerts pour STE-Formations<br>
 */
public class TrainingException extends Exception {

  private static final long serialVersionUID = 1L;
  public TrainingException(String arg0) {
    super(arg0);
  }
  public TrainingException(Throwable cause) {
    super(cause);
  }
}
```

school.client.commons.NetFactory

```
package school.client.commons;

import java.net.MalformedURLException;
import java.net.URL;

/**
 * Fabrique abstraite permettant de récupérer une URL utilisant le protocole
 * adapté à la récupération de dossier de cours.
 *
 * @author boogaerts
 *
 */
public interface NetFactory {

 /**
  * Retourne une {@link URL} vers un dossier de cours en fonction du nom de
  * cours passé en paramètre.
  *
  * @param lessonName
  *       le nom du cours
  * @return l'URL
  * @throws MalformedURLException
  */
 URL getURL(String lessonName) throws MalformedURLException;
}
```

school.client.Main

```
package school.client;

import school.client.commons.Factory;
import school.client.gui.MainWindow;

public class Main {

 public static void main(String[] args) {
  final MainWindow window;

  Factory factory = Factory.getInstance();
  window = factory.getMainWindow();

  javax.swing.SwingUtilities.invokeLater(new Runnable() {
   public void run() {
    createAndShowGUI(window);
   }
  });
 }
 static void createAndShowGUI(MainWindow window) {
  window.setVisible(true);
 }
}
```

school.client.data.DataFactoryImpl

```
package school.client.data;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;

import school.client.commons.Curriculum;
import school.client.commons.DataFactory;
import school.client.commons.Lesson;
import school.client.commons.School;
import school.client.commons.Training;
import school.client.commons.TrainingException;

public class DataFactoryImpl implements DataFactory {

 private static DataFactoryImpl instance;
 private HashMap<String, SchoolImpl> schoolMap;
```

```
 private DataFactoryImpl() {
  super();
 }
 Lesson getNewLesson(String name) throws TrainingException {
  return new LessonImpl(name);
 }
 Training getNewTraining(String trainingName, Curriculum lessonNamesList)
   throws TrainingException {
  return new TrainingImpl(trainingName, lessonNamesList);
 }
 @Override
 public School getSchool(String schoolName,
   Map<String, Curriculum> curriculumMap) {
  SchoolImpl school = this.getSchoolMap().get(schoolName);
  if(school == null){
   List<Training> trainingList = new ArrayList<Training>();
   for( Entry<String, Curriculum> entry : curriculumMap.entrySet()){
    try {
     trainingList.add(this.getNewTraining(entry.getKey(), entry.getValue()));
    } catch (TrainingException e) {
     e.printStackTrace();
    }
   }
   school = new SchoolImpl(schoolName,trainingList);
   this.getSchoolMap().put(schoolName, school);
  }
  return school;
 }
 private HashMap<String, SchoolImpl> getSchoolMap() {
  if (schoolMap == null){
   this.schoolMap = new HashMap<String, SchoolImpl>();
  }
  return schoolMap;
 }
 public static DataFactoryImpl getInstance() {
  if (instance == null){
   instance = new DataFactoryImpl();
  }
  return instance;
 }
}
```

school.client.data.SchoolImpl

```
package school.client.data;

import java.util.Enumeration;
import java.util.Hashtable;
import java.util.List;

import school.client.commons.School;
import school.client.commons.Training;
import school.client.commons.TrainingException;

/**
 * Ecole gérant l'accès à des dossiers de cours.
 *
 * <pre>
 * Propriétés surchargées :
 * - Formations : DosFormation[*]
 * </pre>
 *
 * @author Yannick Boogaerts pour STE-Formations <br>
 * @see School
 * @see Training
 */
public class SchoolImpl implements School {

 /**
  * Le nom de l'école
  */
 private String name;
 /**
  * ensemble des formations de l'école.
  */
```

```
private Hashtable<String, Training> trainingMap;
/**
 * Construit une école.
 *
 * @param schoolName
 *     nom de l'école
 * @param trainingList
 *     la liste des formations de l'école
 */
public SchoolImpl(String schoolName, List<Training> trainingList) {
 super();
 this.trainingMap = new Hashtable<String, Training>();
 if (trainingMap != null) {
  for (Training t : trainingList) {
   this.trainingMap.put(t.getName(), t);
  }
 }
 this.name = schoolName;
}
@Override
public Training getTraining(String trainingName) throws TrainingException {
 return this.trainingMap.get(trainingName);
}
@Override
public Enumeration<String> getTrainingNameEnum() {
 return this.trainingMap.keys();
}
@Override
public String getName() {
 return this.name;
}
@Override
public String toString() {
 return this.getName();
}
}
```

school.client.data.TrainingImpl

```
package school.client.data;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.util.Enumeration;
import java.util.Hashtable;

import school.client.commons.Curriculum;
import school.client.commons.Factory;
import school.client.commons.Lesson;
import school.client.commons.Training;
import school.client.commons.TrainingException;

public class TrainingImpl implements Training {

 /**
  * Ensemble des noms de dossier de cours de la formation.
  */
 private Hashtable<String, Lesson> curriculum = new Hashtable<String, Lesson>();
 /**
  * Nom de la formation.
  */
 private String name;
 private PropertyChangeSupport propertyChangeSupport = new PropertyChangeSupport(
   this);
 /**
  * Instancie une nouvelle {@link Training} en l'initialisant ses propriétés
  * passées en paramètres.
  *
  * @param name
  *     nom de la foramtion
  * @param curriculum
  *     ensemble des noms de cours du curriculum
  * @throws TrainingException
  *      si le nom de la formation ou un élément du paramètre curriculum
  *      est null ou est composé uniquement de caractères d'espacement.
```

```
 */
public TrainingImpl(String name, Curriculum curriculum)
  throws TrainingException {
 this.valideName(name);
 this.name = name;

 for (String lessonName : curriculum) {
  this.addLesson(lessonName);
 }
}
/**
 * Renvoit le nom de la formation.
 *
 * @return le nom de la formation.
 */
public String getName() {
 return this.name;
}
/**
 * Renvoit une énumération des noms de cours du curriculum
 *
 * @return une énumération des noms de cours du curriculum
 */
@Override
public Enumeration<String> getLessonNameEnum() {
 return curriculum.keys();
}
/**
 * Renvoit le dossier de cours correspondant au nom du cours. Si aucun dossier
 * de cours au nom du cours n'est présent dans la référence un nouveau cours
 * avec un dossier erreur est renvoyé.
 *
 * @param lessonName
 *      nom du cours à renvoyer
 * @return dossier de cours correspondant au nom
 * @throws TrainingException
 *       si le nom du cours n'est pas présent dans le curriculum.
 * @throws ClassCastException
 *       si la valeur de reference correspondant au npm n'est pas un
 *       DosCours
 * @see Training#getLesson(String)
 */
public Lesson getLesson(String lessonName) throws TrainingException {
 String key = lessonName.toUpperCase().trim();
 Lesson cours = this.curriculum.get(key);
 if (cours == null)
  throw new TrainingException("cours absent du curriculum");
 return cours;
}
public Lesson addLesson(Training.Callback callback) throws TrainingException {
 return this.addLesson(callback.getName());
}
private Lesson addLesson(String name) throws TrainingException {
 String key;
 Lesson cours;

 this.valideName(name);
 key = name.toUpperCase().trim();

 if (this.curriculum.containsKey(key)) {
  throw new TrainingException(name + " fait déjà partie de la formation");
 } else {
  cours = DataFactoryImpl.getInstance().getNewLesson(name);
  this.curriculum.put(key, cours);

  this.propertyChangeSupport.firePropertyChange("addLesson", null, cours);

  return cours;
 }
}
public Lesson removeLesson(String name) {
 String key = name.toUpperCase().trim();
 Lesson cours = this.curriculum.remove(key);

 this.propertyChangeSupport.firePropertyChange("removeLesson", cours, null);
```

```
 return cours;
}
private void valideName(String name) throws TrainingException {
 if (name == null || name.matches("\\s*"))
  throw new TrainingException("nom incorrect");
}
public String toString() {
 return this.name;
}
@Override
public int hashCode() {
 final int prime = 31;
 int result = 1;
 result = prime * result
   + ((curriculum == null) ? 0 : curriculum.hashCode());
 result = prime * result + ((name == null) ? 0 : name.hashCode());
 return result;
}
@Override
public boolean equals(Object obj) {
 if (this == obj)
  return true;
 if (obj == null)
  return false;
 if (getClass() != obj.getClass())
  return false;
 TrainingImpl other = (TrainingImpl) obj;
 if (curriculum == null) {
  if (other.curriculum != null)
   return false;
 } else if (!curriculum.equals(other.curriculum))
  return false;
 if (name == null) {
  if (other.name != null)
   return false;
 } else if (!name.equals(other.name))
  return false;
 return true;
}
public void addPropertyChangeListener(PropertyChangeListener listener) {
 propertyChangeSupport.addPropertyChangeListener(listener);
}
public void addPropertyChangeListener(String propertyName,
   PropertyChangeListener listener) {
 propertyChangeSupport.addPropertyChangeListener(propertyName, listener);
}
public void removePropertyChangeListener(PropertyChangeListener listener) {
 propertyChangeSupport.removePropertyChangeListener(listener);
}
public void removePropertyChangeListener(String propertyName,
   PropertyChangeListener listener) {
 propertyChangeSupport.removePropertyChangeListener(propertyName, listener);
}
}
```

school.client.data.LessonImpl

```
package school.client.data;

import java.net.MalformedURLException;
import java.net.URL;

import school.client.commons.Factory;
import school.client.commons.Lesson;
import school.client.commons.TrainingException;

public class LessonImpl implements Lesson {

 /**
  * Intitulé du cours
  */
 private String name;
 /**
  * Contenu du cours
  */
```

```java
private URL content;
/**
 * Construit un cours en lui donnant un intitulé. Le contenu est représenté
 * par une URL dont :
 * <ul>
 * <li>le protocole est "book"</li>
 * <li>l'hôte est "localhost"</li>
 * <li>le port est 1443</li>
 * <li>le gestionnaire de flux est un <code>DossierURLStreamHandler</code></li>
 * </ul>
 *
 * @param name
 *     le nom du cours
 * @throws TrainingException
 *      si le nom est null ou s'il est composé uniquement de caractères
 *      d'espacement.
 */
public LessonImpl(String name) throws TrainingException {
 if (name == null || name.matches("\\s*"))
  throw new TrainingException("nom de cours incorrect");
 this.name = name;
 try {
  this.content = Factory.getInstance().getURL(name);

 } catch (MalformedURLException e) {
  throw new TrainingException(e);
 }
}
/**
 * Renvoit l'intitulé du cours
 *
 * @see Lesson#getName()
 */
public String getName() {
 return name;
}
/**
 * Renvoit le contenu du cours.
 *
 * @see Lesson#getContent()
 */
public URL getContent() {
 return content;
}
/**
 * Deux cours sont égaux si leur intitulé et leur contenu sont égaux.
 *
 * @see java.lang.Object#equals(java.lang.Object)
 */
@Override
public boolean equals(Object arg0) {
 return arg0 != null && arg0 instanceof LessonImpl
   && ((LessonImpl) arg0).getName().equals(this.name)
   && ((LessonImpl) arg0).getContent().equals(this.content);
}
/**
 * Renvoit le code de hashage de l'objet.
 *
 * @see java.lang.Object#hashCode()
 */
@Override
public int hashCode() {
 return this.name.hashCode();
}
/**
 * Renvois le DosCours sous forme de String
 *
 * @see java.lang.Object#toString()
 */
@Override
public String toString() {
 return this.name;
}
}
```

school.client.gui.MainWindow

```
package school.client.gui;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Font;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import java.awt.event.ActionListener;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.io.IOException;
import java.net.URL;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JEditorPane;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSplitPane;
import javax.swing.JTree;
import javax.swing.border.TitledBorder;
import javax.swing.event.TreeSelectionListener;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;
import javax.swing.tree.TreeModel;
import javax.swing.tree.TreePath;
import school.client.commons.Training;
import school.client.gui.SchoolObject.Type;

/**
 * Panel d'affichage de l'arborescence Ecole-Formation_cours et du countenu des
 * cours. Le panel contient deux boutons pour géré l'ajout et la suppression de
 * cours.
 *
 * <pre>
 * Structure du component :
 * PanelDossier
 *  +-- JSplitPane
 *    +-- JPanel (gauche)
 *    : +-- JScrollPane (Haut)
 *    : : +-- JTree
 *    : +-- JButton &quot;Ajouter&quot; (bas gauche)
 *    : +-- JButton &quot;Supprimer (bas droite)
 *    +-- JScrollPane (droite)
 *     +-- jEditorPane
 * </pre>
 *
 * @author Yannick Boogaerts pour STE-Formations<br>
 */
public class MainWindow extends JFrame implements PropertyChangeListener {

 private static final long serialVersionUID = -4778723881440423905L;
 private JSplitPane jSplitPane = null;
 private JScrollPane jScrollPaneCours = null;
 private JEditorPane jEditorPane = null;
 private JPanel jPanelListe = null;
 private JScrollPane jScrollPane = null;
 private JTree jTree = null;
 private JButton jButtonadd = null;
 private JButton jButtonSupp = null;
 private JPanel jPanelLogin = null;
 private JButton jButtonLogin = null;
 private JLabel jLabelUser = null;

 /**
  * This is the default constructor
  */
 public MainWindow() {
  super();
  initialize();
 }
```

```
public void addJButtonAddListener(ActionListener listener) {
 this.getJButtonadd().addActionListener(listener);
}
public void removeJButtonAddListener(ActionListener listener) {
 this.getJButtonadd().removeActionListener(listener);
}
public void addJButtonRemoveListener(ActionListener listener) {
 this.getJButtonSupp().addActionListener(listener);
}
public void removeJButtonRemoveListener(ActionListener listener) {
 this.getJButtonSupp().removeActionListener(listener);
}
public void addJButtonLoginListener(ActionListener listener) {
 this.getJButtonLogin().addActionListener(listener);
}
public void removeJButtonLoginListener(ActionListener listener) {
 this.getJButtonLogin().removeActionListener(listener);
}
public void addTreeSelectionListener(TreeSelectionListener listener) {
 this.getJTree().addTreeSelectionListener(listener);
}
public void removeTreeSelectionListener(TreeSelectionListener listener) {
 this.getJTree().removeTreeSelectionListener(listener);
}
public void setTreeModel(TreeModel model) {
 this.getJTree().setModel(model);
}
public void updateText(String text, String contentType) {
 this.getJEditorPane().setContentType(contentType);
 this.getJEditorPane().setText(text);
}
private void initialize() {
 this.setLayout(new BorderLayout());
 this.setSize(461, 334);
 this.setContentPane(getJSplitPane());
 this.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
}
private JSplitPane getJSplitPane() {
 if (jSplitPane == null) {
  jSplitPane = new JSplitPane();
  jSplitPane.setRightComponent(getJScrollPaneCours());
  jSplitPane.setLeftComponent(getJPanelListe());
 }
 return jSplitPane;
}
private JScrollPane getJScrollPaneCours() {
 if (jScrollPaneCours == null) {
  jScrollPaneCours = new JScrollPane();
  jScrollPaneCours.setViewportView(getJEditorPane());
 }
 return jScrollPaneCours;
}
private JEditorPane getJEditorPane() {
 if (jEditorPane == null) {
  jEditorPane = new JEditorPane();
 }
 return jEditorPane;
}
private JPanel getJPanelListe() {
 if (jPanelListe == null) {
  GridBagConstraints gridBagConstraints4 = new GridBagConstraints();
  gridBagConstraints4.gridx = 0;
  gridBagConstraints4.gridwidth = 2;
  gridBagConstraints4.anchor = GridBagConstraints.WEST;
  gridBagConstraints4.fill = GridBagConstraints.HORIZONTAL;
  gridBagConstraints4.gridy = 3;
  GridBagConstraints gridBagConstraints11 = new GridBagConstraints();
  gridBagConstraints11.gridx = 1;
  gridBagConstraints11.fill = java.awt.GridBagConstraints.BOTH;
  gridBagConstraints11.gridy = 1;
  GridBagConstraints gridBagConstraints1 = new GridBagConstraints();
  gridBagConstraints1.gridy = 1;
  GridBagConstraints gridBagConstraints = new GridBagConstraints();
  gridBagConstraints.fill = java.awt.GridBagConstraints.BOTH;
  gridBagConstraints.gridx = 0;
```

```
    gridBagConstraints.gridy = 0;
    gridBagConstraints.weightx = 1.0;
    gridBagConstraints.gridwidth = 2;
    gridBagConstraints.weighty = 1.0;
    jPanelListe = new JPanel();
    jPanelListe.setLayout(new GridBagLayout());
    jPanelListe.add(getJScrollPane(), gridBagConstraints);
    jPanelListe.add(getJButtonadd(), gridBagConstraints1);
    jPanelListe.add(getJButtonSupp(), gridBagConstraints11);
    jPanelListe.add(getJPanelLogin(), gridBagConstraints4);
   }
   return jPanelListe;
  }
  private JScrollPane getJScrollPane() {
   if (jScrollPane == null) {
    jScrollPane = new JScrollPane(getJTree());
   }
   return jScrollPane;
  }
  private JTree getJTree() {
   if (jTree == null) {
    jTree = new JTree();
    jTree.setCellRenderer(new SecureRenderer());
   }
   return jTree;
  }
  private JButton getJButtonadd() {
   if (jButtonadd == null) {
    jButtonadd = new JButton();
    jButtonadd.setText("Ajouter");
    jButtonadd.setActionCommand("add");
   }
   return jButtonadd;
  }
  private JButton getJButtonSupp() {
   if (jButtonSupp == null) {
    jButtonSupp = new JButton();
    jButtonSupp.setText("Supprimer");
    jButtonSupp.setActionCommand("remove");
   }
   return jButtonSupp;
  }
  public DefaultMutableTreeNode getNodeActiveTraining() {
   DefaultMutableTreeNode noeud;

   TreePath parentPath = jTree.getSelectionPath();
   if (parentPath == null) {
    noeud = null;
   } else {
    noeud = (DefaultMutableTreeNode) (parentPath.getLastPathComponent());
    while (noeud != null
      && !(noeud.getUserObject() instanceof Training)) {
     noeud = (DefaultMutableTreeNode) noeud.getParent();
    }
   }
   return noeud;
  }
  public DefaultTreeModel getTreeModel() {
   return (DefaultTreeModel) this.getJTree().getModel();
  }
  public void setSchoolTreeModel(SchoolTreeModel schoolTreeModel) {
   String title = schoolTreeModel.getSchool().getName();
   title += System.getSecurityManager() != null ? " sécurisée" : "";
   this.setTitle(title);
   this.getJTree().setModel(schoolTreeModel);
   this.getJTree().setSelectionRow(0);
   this.getJTree().setSelectionModel(
     schoolTreeModel.getTreeSelectionModel());
  }
  public void updateText(URL support, String contentType) throws IOException {
   this.getJEditorPane().setContentType(contentType);
   this.getJEditorPane().setPage(support);
  }
  @Override
  public void propertyChange(PropertyChangeEvent evt) {
```

```
 if (evt != null && evt.getSource() instanceof SchoolPresentationModel) {

  if ("login".equals(evt.getPropertyName())) {
   this.updateLog((Boolean) evt.getNewValue());
  } else if ("userName".equals(evt.getPropertyName())) {
   this.getJLabelUser().setText((String) evt.getNewValue());
  } else if ("schoolTreeModel".equals(evt.getPropertyName())) {
   this.setSchoolTreeModel((SchoolTreeModel) evt.getNewValue());
  } else if ("selectedObject".equals(evt.getPropertyName())) {
   this.updateSelectedObject((SchoolObject) evt.getNewValue());
  }
 }
}
private void updateSelectedObject(SchoolObject newValue) {
 if (newValue == null) {
  this.updateText("", "text/txt");
 } else if (newValue.getType() == Type.LESSON) {
  try {
   this.updateText(newValue.getLesson().getContent(), "text/html");
  } catch (IOException e) {
   e.printStackTrace();
  }
 } else {
  this.updateText(newValue.getLastName(), "text/txt");
 }
}
private void updateLog(Boolean login) {
 if (login) {
  this.getJButtonLogin().setActionCommand("logout");
  this.getJButtonLogin().setText("logout");
 } else {
  this.getJButtonLogin().setActionCommand("login");
  this.getJButtonLogin().setText("login");
 }

}
private JPanel getJPanelLogin() {
 if (jPanelLogin == null) {
  GridBagConstraints gridBagConstraints3 = new GridBagConstraints();
  gridBagConstraints3.gridx = 1;
  gridBagConstraints3.anchor = GridBagConstraints.EAST;
  gridBagConstraints3.fill = GridBagConstraints.HORIZONTAL;
  gridBagConstraints3.ipadx = 10;
  gridBagConstraints3.insets = new Insets(0, 10, 0, 0);
  gridBagConstraints3.gridy = 0;
  GridBagConstraints gridBagConstraints2 = new GridBagConstraints();
  gridBagConstraints2.gridx = -1;
  gridBagConstraints2.anchor = GridBagConstraints.WEST;
  gridBagConstraints2.fill = GridBagConstraints.NONE;
  gridBagConstraints2.gridy = -1;
  jPanelLogin = new JPanel();
  jPanelLogin.setLayout(new GridBagLayout());
  jPanelLogin.setBorder(BorderFactory.createTitledBorder(null,
    "Utilisateur", TitledBorder.DEFAULT_JUSTIFICATION,
    TitledBorder.DEFAULT_POSITION, new Font("Dialog",
      Font.BOLD, 12), new Color(51, 51, 51)));
  jPanelLogin.add(getJButtonLogin(), gridBagConstraints2);
  jPanelLogin.add(getJLabelUser(), gridBagConstraints3);
 }
 return jPanelLogin;
}
private JLabel getJLabelUser() {
 if (this.jLabelUser == null) {
  jLabelUser = new JLabel();
  jLabelUser.setText("Anonyme");
 }
 return jLabelUser;
}
private JButton getJButtonLogin() {
 if (jButtonLogin == null) {
  jButtonLogin = new JButton();
  jButtonLogin.setActionCommand("login");
  jButtonLogin.setText("Login");
 }
```

```
    return jButtonLogin;
  }
}
```

school.client.gui.SchoolPresentationModel

```
package school.client.gui;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import javax.security.auth.Subject;
import javax.swing.event.TreeSelectionEvent;
import javax.swing.event.TreeSelectionListener;
import school.client.commons.ActionModelListener;
import school.client.commons.ActionCommand;
import school.client.commons.Factory;
import school.client.commons.School;

public class SchoolPresentationModel implements TreeSelectionListener,
  ActionListener {
 private boolean login = false;
 private String userName = "Anonyme";
 private SchoolTreeModel schoolTreeModel = new SchoolTreeModel(Factory
   .getInstance().getSchool("null"));
 private SchoolObject selectedObject;
 private PropertyChangeSupport support = new PropertyChangeSupport(this);
 private ActionModelSupport<ActionCommand> actionSupport = new
ActionModelSupport<ActionCommand>(
   this);

 public void addPropertyChangeListener(PropertyChangeListener listener) {
  support.addPropertyChangeListener(listener);
  listener.propertyChange(new PropertyChangeEvent(this, "login",
    this.login, this.login));
  listener.propertyChange(new PropertyChangeEvent(this, "userName",
    this.userName, this.userName));
  listener.propertyChange(new PropertyChangeEvent(this,
    "schoolTreeModel", this.schoolTreeModel, this.schoolTreeModel));
  listener.propertyChange(new PropertyChangeEvent(this, "selectedObject",
    this.selectedObject, this.selectedObject));
 }
 public void addPropertyChangeListener(String propertyName,
   PropertyChangeListener listener) {
  support.addPropertyChangeListener(propertyName, listener);
  if ("login".equalsIgnoreCase(propertyName))
   listener.propertyChange(new PropertyChangeEvent(this, "login",
     this.login, this.login));
  else if ("userName".equalsIgnoreCase(propertyName))
   listener.propertyChange(new PropertyChangeEvent(this, "userName",
     this.userName, this.userName));
  else if ("schoolTreeModel".equalsIgnoreCase(propertyName))
   listener.propertyChange(new PropertyChangeEvent(this,
     "schoolTreeModel", this.schoolTreeModel,
     this.schoolTreeModel));
  else if ("selectedObject".equalsIgnoreCase(propertyName))
   listener.propertyChange(new PropertyChangeEvent(this, "user",
     this.selectedObject, this.selectedObject));
 }
 public void removePropertyChangeListener(PropertyChangeListener listener) {
  support.removePropertyChangeListener(listener);
 }
 public void removePropertyChangeListener(String propertyName,
   PropertyChangeListener listener) {
  support.removePropertyChangeListener(propertyName, listener);
 }
 public void addActionModelListener(ActionModelListener<ActionCommand> listener) {
  actionSupport.addActionModelListener(listener);
 }
 public void addActionModelListener(ActionCommand action,
   ActionModelListener<ActionCommand> listener) {
  actionSupport.addActionModelListener(action, listener);
 }
```

```
public void removeActionModelListener(ActionModelListener<ActionCommand> listener) {
 actionSupport.removeActionModelListener(listener);
}
public void removeActionModelListener(ActionCommand action,
  ActionModelListener<ActionCommand> listener) {
 actionSupport.removeActionModelListener(action, listener);
}
public boolean isLog() {
 return login;
}
public void setlogin(boolean log, String userName) {
 boolean oldlogin = this.login;
 String oldUserName = this.userName;
 this.login = log;
 this.userName = userName;
 this.support.firePropertyChange("login", oldlogin, this.login);
 this.support.firePropertyChange("userName", oldUserName, this.userName);
}
public School getSchool() {
 return schoolTreeModel.getSchool();
}
public void setSchool(School school) {
 SchoolTreeModel old = this.schoolTreeModel;
 this.schoolTreeModel = new SchoolTreeModel(school);
 this.support.firePropertyChange("schoolTreeModel", old, this.schoolTreeModel);
}
public SchoolObject getSelectedObject() {
 return selectedObject;
}
public void setSelectedObject(SchoolObject selectedObject) {
 SchoolObject old = this.selectedObject;
 this.selectedObject = selectedObject;
 this.support.firePropertyChange("selectedObject", old,
   this.selectedObject);
}
@Override
public void valueChanged(TreeSelectionEvent e) {
 Object[] path = e.getPath().getPath();
 String param[] = new String[path.length];
 for (int i = 0; i < path.length; i++) {
  param[i] = path[i].toString();
 }
 this.actionSupport.fireActionModel(ActionCommand.SELECTED_OBJECT_CHANGE,
   param);
}
@Override
public void actionPerformed(ActionEvent e) {
 if (e != null) {
  if ("add".equals(e.getActionCommand())) {
   this.actionSupport.fireActionModel(ActionCommand.ADD_LESSON, null);
  } else if ("remove".equals(e.getActionCommand())) {
   this.actionSupport.fireActionModel(ActionCommand.REMOVE_LESSON, null);
  } else if ("login".equals(e.getActionCommand())) {
   this.actionSupport.fireActionModel(ActionCommand.LOGIN, null);
  } else if ("logout".equals(e.getActionCommand())) {
   this.actionSupport.fireActionModel(ActionCommand.LOGOUT, null);
  }
 }
}
```

school.client.gui.ActionModelSupport

```
package school.client.gui;

import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;
import school.client.commons.ActionModelEvent;
import school.client.commons.ActionModelListener;

public class ActionModelSupport<A extends Enum> {

 private Set<ActionModelListener<A>> allAction;
```

```
private Map<A, Set<ActionModelListener<A>>> byAction;
private Object source;

public ActionModelSupport(Object source) {
 super();
 if (source == null) {
  throw new NullPointerException();
 }
 this.source = source;
}
public synchronized void addActionModelListener(A action,
  ActionModelListener<A> listener) {
 if (action != null && listener != null) {
  Set<ActionModelListener<A>> listenerSet = this.getByAction().get(
    action);
  if (listenerSet == null) {
   listenerSet = new HashSet<ActionModelListener<A>>();
   this.getByAction().put(action, listenerSet);
  }
  listenerSet.add(listener);
 }
}
public synchronized void removeActionModelListener(A action,
  ActionModelListener<A> listener) {
 if (action != null && listener != null) {
  Set<ActionModelListener<A>> listenerSet = this.getByAction().get(
    action);
  if (listenerSet != null) {
   listenerSet.remove(listener);
  }
 }
}
public synchronized void addActionModelListener(
  ActionModelListener<A> listener) {
 if (listener != null) {
  this.allAction.add(listener);
 }
}
public synchronized void removeActionModelListener(
  ActionModelListener<A> listener) {
 if (listener != null) {
   this.allAction.remove(listener);
 }
}
public void fireActionModel(A command, Object param){
 this.fireActionModel(new ActionModelEvent<A>(this.source,command,param));
}
@SuppressWarnings("unchecked")
public void fireActionModel(ActionModelEvent<A> event){
 Object[] tab;
 if (event != null) {
  Set<ActionModelListener<A>> listenerSet = this.getByAction().get(
    event.getCommand());
  if (listenerSet != null) {
   tab = listenerSet.toArray();
   for( Object li : tab){
    ((ActionModelListener)li).actionModelPerformed(event);
   }
  }
  tab = this.getAllAction().toArray();
  for( Object li : tab){
   ((ActionModelListener)li).actionModelPerformed(event);
  }
 }
}
private Set<ActionModelListener<A>> getAllAction() {
 if (this.allAction == null) {
  this.allAction = new HashSet<ActionModelListener<A>>();
 }
 return this.allAction;
}

private Map<A, Set<ActionModelListener<A>>> getByAction() {
 if (this.byAction == null) {
  this.byAction = new HashMap<A, Set<ActionModelListener<A>>>();
```

```
    }
    return this.byAction;
  }
}
```

school.client.gui.LoginDialog

```
package school.client.gui;

import java.awt.Frame;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.swing.JDialog;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JPasswordField;
import javax.swing.JTextField;

public class LoginDialog {

 private JLabel labelLogin;
 private JLabel labelPass;
 private JTextField fieldLogin;
 private JPasswordField fieldPass;
 private JDialog dialogLogin;
 private JPanel panelLogin;
 private Frame parent;
 private JOptionPane options;

 public LoginDialog(Frame parent) {
  super();
  this.parent = parent;
 }
 public static boolean showLogin(Frame parent, NameCallback name,
   PasswordCallback pass) {
  boolean validInput = false;
  if (name != null || pass != null) {
   LoginDialog loginDialog = new LoginDialog(parent);
   loginDialog.getLabelLogin().setText(
     name != null ? name.getPrompt() : "");
   loginDialog.getLabelLogin().setVisible(name != null);
   loginDialog.getFieldLogin().setVisible(name != null);
   loginDialog.getLabelPass().setText(
     pass != null ? pass.getPrompt() : "");
   loginDialog.getLabelPass().setVisible(pass != null);
   loginDialog.getFieldPass().setVisible(pass != null);

   validInput = loginDialog.askLogin();
   if (name != null) {
    if (validInput) {
     name.setName(loginDialog.getFieldLogin().getText());
    }
    loginDialog.getFieldLogin().setText("");
   }
   if (pass != null) {
    if (validInput) {
     pass.setPassword(loginDialog.getFieldPass().getPassword());
    }
    loginDialog.getFieldPass().setText("");

   }
  }
  return validInput;
 }
 private boolean askLogin() {
  Object reponse;
  this.getDialogLogin().setVisible(true);

  reponse = this.getOptions().getValue();
  return reponse != null
    && ((Integer) reponse).intValue() == JOptionPane.OK_OPTION;
 }
```

```
 private JDialog getDialogLogin() {
  if (this.dialogLogin == null) {
   this.dialogLogin = getOptions().createDialog(this.parent, "login");
   this.dialogLogin.setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
  }
  return dialogLogin;
 }
 private JOptionPane getOptions() {
  if (this.options == null) {
   this.options = new JOptionPane(this.getPanelLogin(),
     JOptionPane.QUESTION_MESSAGE, JOptionPane.OK_CANCEL_OPTION);
  }
  return this.options;
 }
 private JPanel getPanelLogin() {
  if (this.panelLogin == null) {
   this.panelLogin = new JPanel();
   this.panelLogin.setLayout(new GridBagLayout());
   GridBagConstraints gbc = new GridBagConstraints();

   gbc.gridx = 1;
   gbc.gridy = 1;
   gbc.gridwidth = 1;
   this.panelLogin.add(this.getLabelLogin(), gbc);
   gbc.gridy = 2;
   this.panelLogin.add(this.getLabelPass(), gbc);
   gbc.gridx = 2;
   gbc.gridy = 1;
   gbc.gridwidth = GridBagConstraints.REMAINDER;
   gbc.fill = GridBagConstraints.HORIZONTAL;
   this.panelLogin.add(this.getFieldLogin(), gbc);
   gbc.gridy = 2;
   this.panelLogin.add(this.getFieldPass(), gbc);
  }
  return this.panelLogin;
 }
 private JLabel getLabelLogin() {
  if (this.labelLogin == null) {
   this.labelLogin = new JLabel("login : ");
  }
  return labelLogin;
 }
 private JLabel getLabelPass() {
  if (this.labelPass == null) {
   this.labelPass = new JLabel("Mot de passe : ");
  }
  return labelPass;
 }
 private JTextField getFieldLogin() {
  if (this.fieldLogin == null) {
   this.fieldLogin = new JTextField(20);
  }
  return fieldLogin;
 }
 private JPasswordField getFieldPass() {
  if (this.fieldPass == null) {
   this.fieldPass = new JPasswordField(20);
  }
  return fieldPass;
 }
}
```

school.client.gui.SchoolTreeModel

```
package school.client.gui;

import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.util.Enumeration;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;
import javax.swing.tree.DefaultTreeSelectionModel;
import javax.swing.tree.TreePath;
import javax.swing.tree.TreeSelectionModel;
import school.client.commons.School;
```

```
import school.client.commons.Training;

public class SchoolTreeModel extends DefaultTreeModel {

 private School school;
 private TreeSelectionModel treeSelectionModel;

 public SchoolTreeModel(School school) {
  super(new DefaultMutableTreeNode(school.getName()));
  this.school = school;
  this.initRoot(school);
 }
 public void setSchool(School school) {
  this.school = school;
  this.initRoot(school);
 }
 private void initRoot(School school) {
  Enumeration<String> trainingEnum, lessonEnum;
  String trainingName, lessonName;
  DefaultMutableTreeNode root, noeudF, noeudC;

  root = new DefaultMutableTreeNode(school.getName());
  trainingEnum = school.getTrainingNameEnum();
  while (trainingEnum.hasMoreElements()) {
   trainingName = (String) trainingEnum.nextElement();
   try {
    noeudF = new TrainingTreeNode(school.getTraining(trainingName));
    lessonEnum = school.getTraining(trainingName).getLessonNameEnum();
    while (lessonEnum.hasMoreElements()) {
     lessonName = (String) lessonEnum.nextElement();
     try {
      noeudC = new DefaultMutableTreeNode(school.getTraining(
        trainingName).getLesson(lessonName));
     } catch (Exception e) {
      noeudC = new DefaultMutableTreeNode(lessonName);
      e.printStackTrace();
     }
     noeudF.add(noeudC);
    }
   } catch (Exception e) {
    noeudF = new DefaultMutableTreeNode(trainingName);
    e.printStackTrace();
   }
   root.add(noeudF);
   this.setRoot(root);
  }
 }
 private class TrainingTreeNode extends DefaultMutableTreeNode implements
   PropertyChangeListener {

  private static final long serialVersionUID = 1L;

  public TrainingTreeNode(Training training) {
   super(training);
   training.addPropertyChangeListener(this);
  }
  @Override
  public void propertyChange(PropertyChangeEvent evt) {
   DefaultMutableTreeNode fils = null;
   TreePath newTreePath;
   Enumeration<?> enumeration;
   if (evt != null && this.getUserObject().equals(evt.getSource())) {
    if ("addLesson".equals(evt.getPropertyName())) {
     fils = new DefaultMutableTreeNode(evt.getNewValue());
     SchoolTreeModel.this.insertNodeInto(fils, this, this
       .getChildCount());
     newTreePath = new TreePath(SchoolTreeModel.this
       .getPathToRoot(fils));
     SchoolTreeModel.this.getTreeSelectionModel()
       .setSelectionPath(newTreePath);
    } else if ("removeLesson".equals(evt.getPropertyName())) {
     enumeration = this.children();
     while (enumeration.hasMoreElements() && fils == null) {
      fils = (DefaultMutableTreeNode) enumeration
        .nextElement();
```

```
      if (!evt.getOldValue().equals(fils.getUserObject())) {
       fils = null;
      }
     }
     if (fils != null) {
      newTreePath = new TreePath(SchoolTreeModel.this
        .getPathToRoot(fils)).getParentPath();
      SchoolTreeModel.this.removeNodeFromParent(fils);
      SchoolTreeModel.this.getTreeSelectionModel()
        .setSelectionPath(newTreePath);
     }
    }
   }
  }
 }
 public School getSchool() {
  return this.school;
 }
 public TreeSelectionModel getTreeSelectionModel() {
  if (this.treeSelectionModel == null) {
   this.treeSelectionModel = new DefaultTreeSelectionModel();
   //this.treeSelectionModel.setSelectionPath(new TreePath(new Object[](this.root)));
  }
  return treeSelectionModel;
 }
}
```

school.client.gui.SchoolObject

```
package school.client.gui;

import school.client.commons.Lesson;
import school.client.commons.Factory;
import school.client.commons.School;
import school.client.commons.Training;
import school.client.commons.TrainingException;

public class SchoolObject {

 private School school;
 private Training training;
 private Lesson lesson;
 private String lastName;
 private Type type;

 public enum Type {
  SCHOOL, SCHOOL_NAME, TRAINING, TRAINING_NAME, LESSON, LESSON_NAME,
 }
 public SchoolObject(String[] path) {
  super();
  if (path == null || path.length == 0) {
   throw new IllegalArgumentException(
     "school == null && lastName == null");
  } else {
   try {
    this.type = Type.SCHOOL_NAME;
    this.lastName = path[0];
    this.school = Factory.getInstance().getSchool(path[0]);
    this.type = Type.SCHOOL;
    if (path.length > 1) {
     this.type = Type.TRAINING_NAME;
     this.lastName = path[1];
     this.training = this.school.getTraining(path[1]);
     this.type = Type.TRAINING;
     if (path.length > 2) {
      this.type = Type.LESSON_NAME;
      this.lastName = path[2];
      this.lesson = this.training.getLesson(path[2]);
      this.training = this.school.getTraining(path[1]);
      this.type = Type.LESSON;
     }
    }
   } catch (Exception e) {
   }
  }
```

```
}
public School getSchool() {
 return school;
}
public Training getTraining() {
 return training;
}
public Lesson getLesson() {
 return lesson;
}
public String getLastName() {
 return lastName;
}
public Type getType() {
 return type;
}
@Override
public int hashCode() {
 final int prime = 31;
 int result = 1;
 result = prime * result + ((lesson == null) ? 0 : lesson.hashCode());
 result = prime * result
   + ((lastName == null) ? 0 : lastName.hashCode());
 result = prime * result + ((school == null) ? 0 : school.hashCode());
 result = prime * result
   + ((training == null) ? 0 : training.hashCode());
 result = prime * result + ((type == null) ? 0 : type.hashCode());
 return result;
}
@Override
public boolean equals(Object obj) {
 if (this == obj)
  return true;
 if (obj == null)
  return false;
 if (getClass() != obj.getClass())
  return false;
 SchoolObject other = (SchoolObject) obj;
 if (lesson == null) {
  if (other.lesson != null)
   return false;
 } else if (!lesson.equals(other.lesson))
  return false;
 if (lastName == null) {
  if (other.lastName != null)
   return false;
 } else if (!lastName.equals(other.lastName))
  return false;
 if (school == null) {
  if (other.school != null)
   return false;
 } else if (!school.equals(other.school))
  return false;
 if (training == null) {
  if (other.training != null)
   return false;
 } else if (!training.equals(other.training))
  return false;
 if (type == null) {
  if (other.type != null)
   return false;
 } else if (!type.equals(other.type))
  return false;
 return true;
}
}
```

school.client.gui.SecureRenderer

```
package school.client.gui;

import java.awt.Component;
import javax.swing.ImageIcon;
import javax.swing.JTree;
import javax.swing.tree.DefaultTreeCellRenderer;
```

```
public class SecureRenderer extends DefaultTreeCellRenderer {
 ImageIcon secureIcon = createImageIcon("cadenas.jpeg");

 public Component getTreeCellRendererComponent(JTree tree, Object value,
   boolean sel, boolean expanded, boolean leaf, int row,
   boolean hasFocus) {

  super.getTreeCellRendererComponent(tree, value, sel, expanded,
    leaf, row, hasFocus);
  if (leaf) {
   if ((value instanceof String)) {
    setIcon(secureIcon);
   }
  }
  return this;
 }
 private ImageIcon createImageIcon(String path) {
  java.net.URL imgURL = this.getClass().getResource(path);
  if (imgURL != null) {
   return new ImageIcon(imgURL);
  } else {
   System.err.println("Coùldn't find file: " + path);
   return null;
  }
 }
}
```

school.client.action.ActionFactoryImpl

```
package school.client.action;

import school.client.commons.ActionFactory;
import school.client.commons.ActionModelListener;
import school.client.commons.ActionCommand;

public class ActionFactoryImpl implements ActionFactory {

 @Override
 public ActionModelListener<ActionCommand> getAddLessonAction() {
  return new AddLessonAction();
 }
 @Override
 public ActionModelListener<ActionCommand> getSchoolVisitor() {
  return new SchoolVisitor();
 }
 @Override
 public ActionModelListener<ActionCommand> getRemoveLessonAction() {
  return new RemoveLessonAction();
 }
 @Override
 public ActionModelListener<ActionCommand> getLoginAction() {
  return new LoginAction();
 }
}
```

school.client.action.SchoolVisitor

```
package school.client.action;

import school.client.commons.ActionModelEvent;
import school.client.commons.ActionModelListener;
import school.client.commons.ActionCommand;
import school.client.gui.SchoolObject;
import school.client.gui.SchoolPresentationModel;

public class SchoolVisitor implements ActionModelListener<ActionCommand> {
 public SchoolVisitor() {
  super();
 }

 @Override
 public void actionModelPerformed(ActionModelEvent<ActionCommand> event) {
  SchoolPresentationModel source;
  if (event != null
    && event.getSource() instanceof SchoolPresentationModel
```

```
    && event.getCommand() == ActionCommand.SELECTED_OBJECT_CHANGE) {
   source = (SchoolPresentationModel) event.getSource();
   source.setSelectedObject(new SchoolObject((String[]) event
     .getParam()));
  }
 }
}
```

school.client.action.AddLessonAction

```
package school.client.action;

import javax.swing.JOptionPane;
import school.client.commons.ActionModelEvent;
import school.client.commons.ActionModelListener;
import school.client.commons.ActionCommand;
import school.client.commons.TrainingException;
import school.client.commons.Training.Callback;
import school.client.gui.SchoolObject;
import school.client.gui.SchoolPresentationModel;

public class AddLessonAction implements ActionModelListener<ActionCommand> {

 @Override
 public void actionModelPerformed(ActionModelEvent<ActionCommand> event) {
  SchoolPresentationModel source;
  SchoolObject path;
  if (event != null
    && event.getSource() instanceof SchoolPresentationModel
    && event.getCommand() == ActionCommand.ADD_LESSON) {
   source = (SchoolPresentationModel) event.getSource();
   path = source.getSelectedObject();
   if(path != null && path.getTraining() != null ){
    try {
     path.getTraining().addLesson(new Callback() {

      @Override
      public String getName() {
       return JOptionPane.showInputDialog(null,
         "Nom du cours");
      }
     });
    } catch (TrainingException e) {
     JOptionPane.showMessageDialog(null, e.getMessage());
    }
   }
  }
 }
}
```

school.client.action.RemoveLessonAction

```
package school.client.action;

import javax.swing.JOptionPane;

import school.client.commons.ActionModelEvent;
import school.client.commons.ActionModelListener;
import school.client.commons.ActionCommand;
import school.client.commons.TrainingException;
import school.client.gui.SchoolObject;
import school.client.gui.SchoolPresentationModel;
import school.client.gui.SchoolObject.Type;

public class RemoveLessonAction implements ActionModelListener<ActionCommand> {

 @Override
 public void actionModelPerformed(ActionModelEvent<ActionCommand> event) {
  SchoolPresentationModel source;
  SchoolObject path;
  if (event != null && event.getSource() instanceof SchoolPresentationModel
    && event.getCommand() == ActionCommand.REMOVE_LESSON) {
   source = (SchoolPresentationModel) event.getSource();
   path = source.getSelectedObject();
   if (path != null
     && (path.getType() == Type.LESSON || path.getType() == Type.LESSON_NAME)) {
```

```
        try {
         path.getTraining().removeLesson(path.getLastName());
        } catch (TrainingException e) {
         JOptionPane.showMessageDialog(null, e.getMessage());
        }
       }
      }
     }
    }
```

school.client.action.LoginAction

```
package school.client.action;

import javax.security.auth.callback.NameCallback;
import school.client.commons.ActionModelEvent;
import school.client.commons.ActionModelListener;
import school.client.commons.ActionCommand;
import school.client.gui.LoginDialog;
import school.client.gui.SchoolPresentationModel;

public class LoginAction implements ActionModelListener<ActionCommand> {

 @Override
 public void actionModelPerformed(ActionModelEvent<ActionCommand> event) {
  SchoolPresentationModel source;
  if (event != null && event.getSource() instanceof SchoolPresentationModel) {
   source = (SchoolPresentationModel) event.getSource();
   if (event.getCommand() == ActionCommand.LOGIN) {
    NameCallback nc = new NameCallback("Nom d'utilisateur");
    // PasswordCallback pc = new PasswordCallback("mot de passe",
    // false);
    if (LoginDialog.showLogin(null, nc, null)) {
     // pc.clearPassword();
     source.setlogin(true, nc.getName());
     source.setSchool(source.getSchool());
    }
   } else if (event.getCommand() == ActionCommand.LOGOUT) {
    source.setlogin(false, "Anonyme");
    source.setSchool(source.getSchool());
   }
  }
 }
}
```

school.curriculum.CurriculumFactoryImpl

```
package school.curriculum;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.util.Arrays;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Map;
import java.util.Properties;
import school.client.commons.Curriculum;
import school.client.commons.CurriculumFactory;

public class CurriculumFactoryImpl implements CurriculumFactory {

 private Hashtable<String, Curriculum> curriculumlMap;

 @Override
 public Map<String, Curriculum> getCurriculumMap() {
  if (this.curriculumlMap == null) {
   this.curriculumlMap = new Hashtable<String, Curriculum>();
   CurriculumImpl cours;
   Properties prop = new Properties();
   try {
    this.LoadProperties(prop, "formations.txt");

    for (String trainingName : prop.stringPropertyNames()) {
     cours = new CurriculumImpl(Arrays.asList(prop.getProperty(
       trainingName).split(",")));
```

```
        this.curriculumlMap.put(trainingName, cours);
      }
    } catch (IOException e) {
     e.printStackTrace();
    }
   }
   return curriculumlMap;
  }
  private void LoadProperties(Properties prop, String path)
    throws IOException {
   InputStream in = CurriculumFactoryImpl.class.getResourceAsStream(path);
   if (in == null)
    throw new FileNotFoundException(path);
   prop.load(in);
  }
  @Override
  public String getSchoolName() {
   return "STE-Formations";
  }
}
```

school.curriculum.CurriculumImpl

```
package school.curriculum;

import java.util.Iterator;
import java.util.List;

import school.client.commons.Curriculum;

public class CurriculumImpl implements Curriculum {

 List<String> lessonList;

 public CurriculumImpl(List<String> lessonList) {
  super();
  if (lessonList==null)throw new NullPointerException("Liste de cours null");
  this.lessonList = lessonList;
 }
 @Override
 public Iterator<String> iterator() {
  return lessonList.iterator();
 }
}
```

school.net.NetFactoryImpl

```
package school.net;

import java.net.MalformedURLException;
import java.net.URL;
import school.client.commons.NetFactory;

public class NetFactoryImpl implements NetFactory {

 @Override
 public URL getURL(String lessonName) throws MalformedURLException {
  return new URL("book", "localhost", 1443, lessonName,
    new BookURLStreamHandler());
 }
}
```

school.net.BookURLStreamHandler

```
package school.net;

import java.io.IOException;
import java.net.URL;
import java.net.URLConnection;
import java.net.URLStreamHandler;

public class BookURLStreamHandler extends URLStreamHandler {

 /**
  * renvoie une DossiersIURLConnection;
  */
 @Override
```

```
 protected URLConnection openConnection(URL u) throws IOException {
  if ("book".equals(u.getProtocol())) {
   return new BookURLConnection(u);
  }
  throw new IOException("Protocol not supported: " + u.getProtocol());
 }
}
```

school.net.BookURLConnection

```
package school.net;

import java.io.IOException;
import java.io.InputStream;
import java.io.PrintStream;
import java.net.Socket;
import java.net.URL;
import java.net.URLConnection;

/**
 * Classe gérant le protocole élémentaire "book".
 */
public class BookURLConnection extends URLConnection {

 final static int defaultPort = 1443;

 private InputStream input;

 /**
  * Construit une {@link BookURLConnection} avec l'URL passée en parmamètre.
  *
  * @param url
  *     l'url
  */
 public BookURLConnection(URL url) {
  super(url);
 }
 /**
  * Crée une connexion à l'hôte et au port référencés par l'URL passé en
  * paramètre lors de la construction de l'instance. Si le port n'est pas
  * précisé dans l'URL, il utilise le port par défaut 1443. Si la connection
  * est établie la méthode initialise la propriété input avec l'inputStream
  * récupéré lors de la connexion et le propriété connected à true.
  */
 @Override
 public void connect() throws IOException {
  URL locURL = super.getURL();
  int port = locURL.getPort();
  if (port == -1) {
   port = BookURLConnection.defaultPort;
  }

  Socket socket = new Socket(locURL.getHost(), port);

  this.input = socket.getInputStream();
  super.connected = true;

  PrintStream out = new PrintStream(socket.getOutputStream());
  out.println("book:" + locURL.getFile());
 }
 /**
  * renvoi le type "text/html" (type de tous les dossiers)
  */
 @Override
 public String getContentType() {
  return "text/html";
 }
 /**
  * retourne l'inputStream de la connexion. la méthode tente d'établir la
  * connection si ce n'est déjà fait.
  */
 @Override
 public synchronized InputStream getInputStream() throws IOException {
  if (!connected)
   this.connect();
```

```
    return input;
  }
}
```

school.server.BookReader

```
package school.server;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.PrintStream;
import java.net.Socket;
import java.net.URL;
import java.util.Arrays;
import java.util.Properties;

/**
 * Thread gérant la réponse à une requête du protocole "book".<br>
 * Un <code>BookReader</code> est définit par :
 * <ul>
 * <li>socket : Socket connecté.</li>
 * <li>mapping : Propeties contenant les nom et chemin vers les dossiers de
 * cours.</li>
 * </ul>
 *
 */
public class BookReader extends Thread {

 private Socket socket;
 private Properties mapping;
 /**
  * Construit un BookReader en initialisant ses propriétés avec les
  * paramèteres.
  *
  * @param socket
  * @param mapping
  */
 public BookReader(Socket socket, Properties mapping) {
  this.socket = socket;
  this.mapping = mapping;
 }
 /**
  * Retourne, via le socket de la classe, le dossier sous forme HTML. La
  * requête doit avoir le format : <code>book:&lt;nom de cours&gt;</code>.<br>
  * Si le nom de cours ne correspond à aucune clé dans le mapping ou que le
  * fichier HTML n'est pas présent, un message d'erreur est renvoyé.
  */
 @Override
 public void run() {
  String[] req;
  String fileName;
  InputStream file;
  URL urlFile = null;

  try {
   OutputStream out = this.socket.getOutputStream();
   try {
    BufferedReader in = new BufferedReader(new InputStreamReader(
      this.socket.getInputStream()));

    req = in.readLine().split(":");

    if (req.length == 2 && req[0].equals("book")) {

     req[1] = req[1].toUpperCase().trim();
     fileName = this.mapping.getProperty(req[1], "Erreur.html");
     urlFile = BookReader.class.getResource(fileName);
     if (urlFile == null) {
      runErreur(out);
     } else {
      file = urlFile.openStream();
```

```
            while (file.available() > 0) {
              out.write(file.read());
            }
            out.close();
          }
        } else {
          runErreur(out);
        }
      } catch (FileNotFoundException e) {
        this.runErreur(out);
      }
    } catch (IOException e) {
    }

  }
  private void runErreur(OutputStream out) {
    PrintStream ps = new PrintStream(out);
    ps.println("aucun Dossier ne répond à la requête");
    try {
      out.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

school.server.BookServer

```
package school.server;

import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.URL;
import java.util.Enumeration;
import java.util.Properties;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.WindowConstants;

/**
 * Classe main permettant l'exécution d'un serveur réseau donnant accès à des
 * dossiers de cours. Un BookServeur est définit par le fichier de propriété
 * "ListeCours.txt" situé dans le même répertoire que la classe. le fichier de
 * propriété à la forme :
 * <ul>
 * <li>clé : nom du cours</li>
 * <li>valeur : nom du fichier HTML présentant le dossier de cours.</li>
 * <ul>
 * les fichiers HTML se situe dans le même repertoire que la classe.
 *
 * @author boogaerts
 */
public class BookServer implements MouseListener {
  private Properties mapping;
  private boolean stop = false;
  private ServerSocket s;
  /**
   * Retourne et initialise une instance de <code>Properties</code> à partir du
   * fichier "ListeCours.txt".
   *
   * @return le mapping vers les fichiers ressources du serveur
   * @throws IOException
   *       si la ListeCours ne peut être chargé.
   */
  public Properties getMapping() throws IOException {
    if (this.mapping == null) {
      this.mapping = new Properties();
      Properties prop = new Properties();
      String libelle, key;
      Enumeration<Object> liste;
      URL url = BookServer.class.getResource("ListeCours.txt");
      if (url == null)
        throw new IOException();
```

```
        prop.load(url.openStream());
        liste = prop.keys();
        while (liste.hasMoreElements()) {
          libelle = (String) liste.nextElement();
          key = libelle.toUpperCase().trim();
          this.mapping.put(key, prop.getProperty(libelle));
        }
      }
      return this.mapping;
    }
    /**
     * Lance le serveur qui écoute sur le port 1443.
     *
     * @param args
     *     pas utilisé.
     */
    public static void main(String[] args) {
      BookServer serveur = new BookServer();
      JFrame ecran = new JFrame("Serveur");
      JButton close = new JButton("Stop");
      close.addMouseListener(serveur);
      ecran.add(close);
      ecran.setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
      ecran.setSize(120, 60);
      ecran.setVisible(true);
      serveur.run();
      ecran.dispose();
    }
    /**
     * Met le seveur en ecoute.
     */
    public void run() {
      try {
        s = new ServerSocket(1443);
        while (!stop) {
          new BookReader(s.accept(), this.getMapping()).start();
        }
      } catch (java.net.SocketException e) {

      } catch (IOException e) {
        e.printStackTrace();
      }
    }
    @Override
    public void mouseClicked(MouseEvent arg0) {
      if (s != null) {
        try {
          s.close();
        } catch (Exception e) {
        }
      }
      this.stop = true;
    }
    @Override
    public void mouseEntered(MouseEvent arg0) {
    }
    @Override
    public void mouseExited(MouseEvent arg0) {
    }
    @Override
    public void mousePressed(MouseEvent arg0) {
    }
    @Override
    public void mouseReleased(MouseEvent arg0) {
    }
}
```